7 Spin Locks and Contention

When writing programs for uniprocessors, it is usually safe to ignore the underlying system's architectural details. Unfortunately, multiprocessor programming has yet to reach that state, and for the time being, it is crucial to understand the underlying machine architecture. The goal of this chapter is to understand how architecture affects performance, and how to exploit this knowledge to write efficient concurrent programs. We revisit the familiar mutual exclusion problem, this time with the aim of devising mutual exclusion protocols that work well with today's multiprocessors.

Any mutual exclusion protocol poses the question: what do you do if you cannot acquire the lock? There are two alternatives. If you keep trying, the lock is called a *spin lock*, and repeatedly testing the lock is called *spinning*, or *busy–waiting*. The `Filter` and `Bakery` algorithms are spin locks. Spinning is sensible when you expect the lock delay to be short. For obvious reasons, spinning makes sense only on multiprocessors. The alternative is to suspend yourself and ask the operating system's scheduler to schedule another thread on your processor, which is sometimes called *blocking*. Because switching from one thread to another is expensive, blocking makes sense only if you expect the lock delay to be long. Many operating systems mix both strategies, spinning for a short time and then blocking. Both spinning and blocking are important techniques. In this chapter, we turn our attention to locks that use spinning.

7.1 Welcome to the Real World

We approach real-world mutual exclusion using the `Lock` interface from the `java.util.concurrent.locks` package. For now, we consider only the two principal

methods: `lock()` and `unlock()`. In most cases, these methods should be used in the following structured way:

```
1  Lock mutex = new LockImpl(...); // lock implementation
2  ...
3  mutex.lock();
4  try {
5    ...          // body
6    } finally {
7      mutex.unlock();
8    }
```

We create a new `Lock` object called `mutex` (Line 1). Because `Lock` is an interface and not a class, we cannot create `Lock` objects directly. Instead, we create an object that *implements* the `Lock` interface. (The java.util.concurrent.locks package includes a number of classes that implement `Lock`, and we provide others in this chapter.) Next, we acquire the lock (Line 3), and enter the critical section, a **`try`** block (Line 4). The **`finally`** block (Line 6) ensures that no matter what, the lock is released when control leaves the critical section. Do not put the `lock()` call inside the **`try`** block, because the `lock()` call might throw an exception before acquiring the lock, causing the **`finally`** block to call `unlock()` when the lock has not actually been acquired.

If we want to implement an efficient `Lock`, why not use one of the algorithms we studied in Chapter 2, such as `Filter` or `Bakery`? One problem with this approach is clear from the space lower bound we proved in Chapter 2: no matter what we do, mutual exclusion using `reads` and `writes` requires space linear in n, the number of threads that may potentially access the location. It gets worse.

Consider, for example, the 2-thread `Peterson` lock algorithm of Chapter 2, presented again in Fig. 7.1. There are two threads, A and B, with IDs either 0 or 1. When thread A wants to acquire the lock, it sets `flag`$[A]$ to *true*, sets `victim` to A, and tests `victim` and `flag`$[1 - A]$. As long as the test fails, the thread spins, repeating the test. Once it succeeds, it enters the critical section, lowering `flag`$[A]$ to *false* as it leaves. We know, from Chapter 2, that the `Peterson` lock provides starvation-free mutual exclusion.

```
1  class Peterson implements Lock {
2    private boolean[] flag = new boolean[2];
3    private int victim;
4    public void lock() {
5      int i = ThreadID.get(); // either 0 or 1
6      int j = 1-i;
7      flag[i] = true;
8      victim = i;
9      while (flag[j] && victim == i) {}; // spin
10   }
11 }
```

Figure 7.1 The `Peterson` class (Chapter 2): the order of reads–writes in Lines 7, 8, and 9 is crucial to providing mutual exclusion.

Suppose we write a simple concurrent program in which each of the two threads repeatedly acquires the `Peterson` lock, increments a shared counter, and then releases the lock. We run it on a multiprocessor, where each thread executes this acquire–increment–release cycle, say, half a million times. On most modern architectures, the threads finish quickly. Alarmingly, however, we may discover that the counter's final value may be slightly off from the expected million mark. Proportionally, the error is probably tiny, but why is there any error at all? Somehow, it must be that both threads are occasionally in the critical section at the same time, even though we have proved that this cannot happen. To quote Sherlock Holmes

> How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?

It must be that our proof fails, not because there is anything wrong with our logic, but because our assumptions about the real world are mistaken.

When programming our multiprocessor, we naturally assumed that read–write operations are atomic, that is, they are linearizable to some sequential execution, or at the very least, that they are sequentially consistent. (Recall that linearizability implies sequential consistency.) As we saw in Chapter 3, sequential consistency implies that there is some global order on all operations in which each thread's operations take effect as ordered by its program. Without calling attention to it at the time, we relied on the assumption that memory is sequentially consistent when proving the `Peterson` lock correct. In particular, mutual exclusion depends on the order of the steps in Lines 7, 8, and 9 of Fig. 7.1. Our proof that the `Peterson` lock provided mutual exclusion implicitly relied on the assumption that any two memory accesses by the same thread, even to separate variables, take effect in program order. (Specifically, it was crucial that B's write to `flag`[B] take effect before its write to `victim` (Eq. 2.3.9) and that A's write to `victim` take effect before its read of `flag`[B] (Eq. 2.3.11).)

Unfortunately, modern multiprocessors typically do not provide sequentially consistent memory, nor do they necessarily guarantee program order among reads–writes by a given thread.

Why not? The first culprits are compilers that reorder instructions to enhance performance. Most programming languages preserve program order for each individual variable, but not across multiple variables. It is therefore possible that the order of writes of `flag`[B] and `victim` by thread B will be reversed by the compiler, invalidating Eq. 2.3.9. A second culprit is the multiprocessor hardware itself. (Appendix B has a much more complete discussion of the multiprocessor architecture issues raised in this chapter.) Hardware vendors make no secret of the fact that writes to multiprocessor memory do not necessarily take effect when they are issued, because in most programs the vast majority of writes do not *need* to take effect in shared memory right away. Thus, on many multiprocessor architectures, writes to shared memory are buffered in a special *write buffer* (sometimes called a *store buffer*), to be written to memory only when needed. If thread A's write to `victim` is delayed in a write buffer, it may arrive in memory only after A reads `flag`[B], invalidating Eq. 2.3.11.

How then does one program multiprocessors given such weak memory consistency guarantees? To prevent the reordering of operations resulting from write buffering, modern architectures provide a special *memory barrier* instruction (sometimes called a *memory fence*) that forces outstanding operations to take effect. It is the programmer's responsibility to know where to insert a memory barrier (e.g., the `Peterson` lock can be fixed by placing a barrier right before each read). Not surprisingly, memory barriers are expensive, about as expensive as an atomic `compareAndSet()` instruction, so we want to minimize their use. In fact, *synchronization instructions* such as `getAndSet()` or `compareAndSet()` described in earlier chapters include a memory barrier on many architectures, as do `reads` and `writes` to `volatile` fields.

Given that barriers cost about as much as synchronization instructions, it may be sensible to design mutual exclusion algorithms directly to use operations such as `getAndSet()` or `compareAndSet()`. These operations have higher consensus numbers than `reads` and `writes`, and they can be used in a straightforward way to reach a kind of consensus on who can and cannot enter the critical section.

7.2 Test-And-Set Locks

The `testAndSet()` operation, with consensus number two, was the principal synchronization instruction provided by many early multiprocessor architectures. This instruction operates on a single memory word (or byte). That word holds a binary value, *true* or *false*. The `testAndSet()` instruction atomically stores *true* in the word, and returns that word's previous value, *swapping* the value *true* for the word's current value. At first glance, this instruction seems ideal for implementing a spin lock. The lock is free when the word's value is *false*, and busy when it is *true*. The `lock()` method repeatedly applies `testAndSet()` to the location until that instruction returns *false* (i.e., until the lock is free). The `unlock()` method simply writes the value *false* to it.

The java.util.concurrent package includes an `AtomicBoolean` class that stores a Boolean value. It provides a `set(`*b*`)` method to replace the stored value with value *b*, and a `getAndSet(`*b*`)` that atomically replaces the current value with *b*, and returns the previous value. The archaic `testAndSet()` instruction is the same as a call to `getAndSet(`*true*`)`. We use the term *test-and-set* in prose to remain compatible with common usage, but we use the expression `getAndSet(`*true*`)` in our code examples to remain compatible with Java. The `TASLock` class shown in Fig. 7.2 shows a lock algorithm based on the `testAndSet()` instruction.

Now consider the alternative to the `TASLock` algorithm illustrated in Fig. 7.3. Instead of performing the `testAndSet()` directly, the thread repeatedly reads the lock until it appears to be free (i.e., until `get()` returns *false*). Only after the lock appears to be free does the thread apply `testAndSet()`. This technique is called *test-and-test-and-set* and the lock a `TTASLock`.

```
1  public class TASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (state.getAndSet(true)) {}
5    }
6    public void unlock() {
7      state.set(false);
8    }
9  }
```

Figure 7.2 The TASLock class.

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {};
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13 }
```

Figure 7.3 The TTASLock class.

Clearly, the TASLock and TTASLock algorithms are equivalent from the point of view of correctness: each one guarantees deadlock-free mutual exclusion. Under the simple model we have been using so far, there should be no difference between these two algorithms.

How do they compare on a real multiprocessor? Experiments that measure the elapsed time for n threads to execute a short critical section invariably yield the results shown schematically in Fig. 7.4. Each data point represents the same amount of work, so in the absence of contention effects, all curves would be flat. The top curve is the TASLock, the middle curve is the TTASLock, and the bottom curve shows the time that would be needed if the threads did not interfere at all. The difference is dramatic: the TASLock performs very poorly, and the TTASLock performance, while substantially better, still falls far short of the ideal.

These differences can be explained in terms of modern multiprocessor architectures. First, a word of caution. Modern multiprocessors encompass a variety of architectures, so we must be careful about overgeneralizing. Nevertheless, (almost) all modern architectures have similar issues concerning caching and locality. The details differ, but the principles remain the same.

For simplicity, we consider a typical multiprocessor architecture in which processors communicate by a shared broadcast medium called a *bus* (like a tiny Ethernet). Both the processors and the memory controller can broadcast on the bus, but

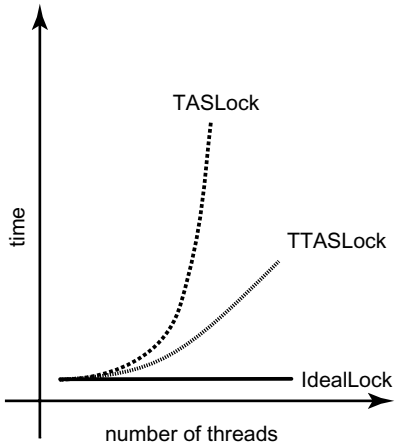

Figure 7.4 Schematic performance of a `TASLock`, a `TTASLock`, and an ideal lock with no overhead.

only one processor (or memory) can broadcast on the bus at a time. All processors (and memory) can listen. Today, bus-based architectures are common because they are easy to build, although they scale poorly to large numbers of processors.

Each processor has a *cache*, a small high-speed memory where the processor keeps data likely to be of interest. A memory access typically requires orders of magnitude more machine cycles than a cache access. Technology trends are not helping: it is unlikely that memory access times will catch up with processor cycle times in the near future, so cache performance is critical to the overall performance of a multiprocessor architecture.

When a processor reads from an address in memory, it first checks whether that address and its contents are present in its cache. If so, then the processor has a *cache hit*, and can load the value immediately. If not, then the processor has a *cache miss*, and must find the data either in the memory, or in another processor's cache. The processor then broadcasts the address on the bus. The other processors *snoop* on the bus. If one processor has that address in its cache, then it responds by broadcasting the address and value. If no processor has that address, then the memory itself responds with the value at that address.

7.3 TAS-Based Spin Locks Revisited

We now consider how the simple `TTASLock` algorithm performs on a shared-bus architecture. Each `getAndSet()` call is broadcast on the bus. Because all threads must use the bus to communicate with memory, these `getAndSet()` calls delay all threads, even those not waiting for the lock. Even worse, the `getAndSet()` call forces other processors to discard their own cached copies of the lock, so

every spinning thread encounters a cache miss almost every time, and must use the bus to fetch the new, but unchanged value. Adding insult to injury, when the thread holding the lock tries to release it, it may be delayed because the bus is monopolized by the spinners. We now understand why the `TASLock` performs so poorly.

Now consider the behavior of the `TTASLock` algorithm while the lock is held by a thread *A*. The first time thread *B* reads the lock it takes a cache miss, forcing *B* to block while the value is loaded into *B*'s cache. As long as *A* holds the lock, *B* repeatedly rereads the value, but hits in the cache every time. *B* thus produces no bus traffic, and does not slow down other threads' memory accesses. Moreover, a thread that releases a lock is not delayed by threads spinning on that lock.

The situation deteriorates, however, when the lock is released. The lock holder releases the lock by writing *false* to the lock variable, which immediately invalidates the spinners' cached copies. Each one takes a cache miss, rereads the new value, and they all (more-or-less simultaneously) call `getAndSet()` to acquire the lock. The first to succeed invalidates the others, who must then reread the value, causing a storm of bus traffic. Eventually, the threads settle down once again to local spinning.

This notion of *local spinning*, where threads repeatedly reread cached values instead of repeatedly using the bus, is an important principle critical to the design of efficient spin locks.

7.4 Exponential Backoff

We now consider how to refine the `TTASLock` algorithm. First, some terminology: *contention* occurs when multiple threads try to acquire a lock at the same time. *High contention* means there are many such threads, and *low contention* means the opposite.

Recall that in the `TTASLock` class, the `lock()` method takes two steps: it repeatedly reads the lock, and when the lock appears to be free, it attempts to acquire the lock by calling `getAndSet(`*true*`)`. Here is a key observation: if some other thread acquires the lock between the first and second step, then, most likely, there is high contention for that lock. Clearly, it is a bad idea to try to acquire a lock for which there is high contention. Such an attempt contributes to bus traffic (making the traffic jam worse), at a time when the thread's chances of acquiring the lock are slim. Instead, it is more effective for the thread to *back off* for some duration, giving competing threads a chance to finish.

For how long should the thread back off before retrying? A good rule of thumb is that the larger the number of unsuccessful tries, the higher the likely contention, and the longer the thread should back off. Here is a simple approach. Whenever the thread sees the lock has become free but fails to acquire it, it backs

off before retrying. To ensure that concurrent conflicting threads do not fall into lock-step, all trying to acquire the lock at the same time, the thread backs off for a random duration. Each time the thread tries and fails to get the lock, it doubles the expected back-off time, up to a fixed maximum.

Because backing off is common to several locking algorithms, we encapsulate this logic in a simple `Backoff` class, shown in Fig. 7.5. The constructor takes these arguments: `minDelay` is the initial minimum delay (it makes no sense for the thread to back off for too short a duration), and `maxDelay` is the final maximum delay (a final limit is necessary to prevent unlucky threads from backing off for much too long). The `limit` field controls the current delay limit. The `backoff()` method computes a random delay between zero and the current limit, and blocks the thread for that duration before returning. It doubles the limit for the next back-off, up to `maxDelay`.

Fig. 7.6 illustrates the `BackoffLock` class. It uses a `Backoff` object whose minimum and maximum back-off durations are governed by the constants `minDelay` and `maxDelay`. It is important to note that the thread backs off only when it fails to acquire a lock that it had immediately before observed to be free. Observing that the lock is held by another thread says nothing about the level of contention.

The `BackoffLock` is easy to implement, and typically performs significantly better than `TASLock` on many architectures. Unfortunately, its performance is sensitive to the choice of `minDelay` and `maxDelay` constants. To deploy this lock on a particular architecture, it is easy to experiment with different values, and to choose the ones that work best. Experience shows, however, that these optimal values are sensitive to the number of processors and their speed, so

```
1  public class Backoff {
2    final int minDelay, maxDelay;
3    int limit;
4    final Random random;
5    public Backoff(int min, int max) {
6      minDelay = min;
7      maxDelay = min;
8      limit = minDelay;
9      random = new Random();
10   }
11   public void backoff() throws InterruptedException {
12     int delay = random.nextInt(limit);
13     limit = Math.min(maxDelay, 2 * limit);
14     Thread.sleep(delay);
15   }
16 }
```

Figure 7.5 The `Backoff` class: adaptive backoff logic. To ensure that concurrently contending threads do not repeatedly try to acquire the lock at the same time, threads back off for a random duration. Each time the thread tries and fails to get the lock, it doubles the expected time to back off, up to a fixed maximum.

```
1  public class BackoffLock implements Lock {
2    private AtomicBoolean state = new AtomicBoolean(false);
3    private static final int MIN_DELAY = ...;
4    private static final int MAX_DELAY = ...;
5    public void lock() {
6      Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
7      while (true) {
8        while (state.get()) {};
9        if (!state.getAndSet(true)) {
10         return;
11       } else {
12         backoff.backoff();
13       }
14     }
15   }
16   public void unlock() {
17     state.set(false);
18   }
19   ...
20 }
```

Figure 7.6 The Exponential Backoff lock. Whenever the thread fails to acquire a lock that became free, it backs off before retrying.

it is not easy to tune the `BackoffLock` class to be portable across a range of different machines.

7.5 Queue Locks

We now explore a different approach to implementing scalable spin locks, one that is slightly more complicated than backoff locks, but inherently more portable. There are two problems with the `BackoffLock` algorithm.

- *Cache-coherence Traffic*: All threads spin on the same shared location causing cache-coherence traffic on every successful lock access (though less than with the `TASLock`).
- *Critical Section Underutilization*: Threads delay longer than necessary, causing the the critical section to be underutilized.

One can overcome these drawbacks by having threads form a *queue*. In a queue, each thread can learn if its turn has arrived by checking whether its predecessor has finished. Cache-coherence traffic is reduced by having each thread spin on a different location. A queue also allows for better utilization of the critical section, since there is no need to guess when to attempt to access it: each thread is notified directly by its predecessor in the queue. Finally, a queue provides first-come-first-served fairness, the same high level of fairness achieved by the `Bakery`

algorithm. We now explore different ways to implement *queue locks*, a family of locking algorithms that exploit these insights.

7.5.1 Array-Based Locks

Figs. 7.7 and 7.8 show the `ALock`,[1] a simple array-based queue lock. The threads share an `AtomicInteger` `tail` field, initially zero. To acquire the lock, each thread atomically increments `tail` (Line 17). Call the resulting value the thread's *slot*. The slot is used as an index into a Boolean `flag` array. If `flag`[*j*] is *true*, then the thread with slot *j* has permission to acquire the lock. Initially, `flag`[0] is *true*. To acquire the lock, a thread spins until the flag at its slot becomes *true* (Line 19). To release the lock, the thread sets the flag at its slot to *false* (Line 23), and sets the flag at the next slot to *true* (Line 24). All arithmetic is modulo *n*, where *n* is at least as large as the maximum number of concurrent threads.

In the `ALock` algorithm, `mySlotIndex` is a *thread-local* variable (see Appendix A). Thread-local variables differ from their regular counterparts in that

```
1  public class ALock implements Lock {
2    ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> (){
3      protected Integer initialValue() {
4        return 0;
5      }
6    };
7    AtomicInteger tail;
8    boolean[] flag;
9    int size;
10   public ALock(int capacity) {
11     size = capacity;
12     tail = new AtomicInteger(0);
13     flag = new boolean[capacity];
14     flag[0] = true;
15   }
16   public void lock() {
17     int slot = tail.getAndIncrement() % size;
18     mySlotIndex.set(slot);
19     while (! flag[slot]) {};
20   }
21   public void unlock() {
22     int slot = mySlotIndex.get();
23     flag[slot] = false;
24     flag[(slot + 1) % size] = true;
25   }
26 }
```

Figure 7.7 Array-based Queue Lock.

1 Most of our lock classes use the initials of their inventors, as explained in Section 7.10

each thread has its its own, independently initialized copy of each variable. Thread-local variables need not be stored in shared memory, do not require synchronization, and do not generate any coherence traffic since they are accessed by only one thread. The value of a thread-local variable is accessed by `get()` and `set()` methods.

The `flag[]` array, on the other hand, is shared. However, contention on the array locations is minimized since each thread, at any given time, spins on its locally cached copy of a single array location, greatly reducing invalidation traffic.

Note that contention may still occur because of a phenomenon called *false sharing*, which occurs when adjacent data items (such as array elements) share a single cache line. A write to one item invalidates that item's cache line, which causes invalidation traffic to processors that are spinning on unchanged but near items that happen to fall in the same cache line. In the example in Fig. 7.8, threads accessing the 8 `ALock` locations may suffer unnecessary invalidations because the locations were all cached in the same two 4-word lines. One way to avoid false sharing is to *pad* array elements so that distinct elements are mapped to distinct cache lines. Padding is easier in low-level languages like C or C++ where the programmer has a direct control over the layout of objects in memory. In the example in Fig. 7.8, we pad the eight original `ALock` locations by increasing the lock array size fourfold, and placing the locations four words apart so that no two locations can fall in the same cache line. (We increment from one location i to the next by computing $4(i + 1) \bmod 32$ instead of $i + 1 \bmod 8$).

7.5.2 The CLH Queue Lock

The `ALock` improves on `BackoffLock` because it reduces invalidations to a minimum, and minimizes the interval between when a lock is freed by one thread and when it is acquired by another. Unlike the `TASLock` and `BackoffLock`, this algorithm guarantees that no starvation occurs, and provides first-come-first-served fairness.

Unfortunately, the `ALock` lock is not space-efficient. It requires a known bound n on the maximum number of concurrent threads, and it allocates an array of that size per lock. Thus, synchronizing L distinct objects requires $O(Ln)$ space, even if a thread accesses only one lock at a time.

We now turn our attention to a different style of queue lock. Fig. 7.9 shows the `CLHLock` class's fields, constructor, and `QNode` class. This class records each thread's status in a `QNode` object, which has a Boolean `locked` field. If that field is *true*, then the corresponding thread has either acquired the lock, or is waiting for the lock. If that field is *false*, then the thread has released the lock. The lock itself is represented as a virtual linked list of `QNode` objects. We use the term "virtual" because the list is implicit: each thread refers to its predecessor through a thread-local `pred` variable. The public `tail` field is an `AtomicReference<QNode>` to the node most recently added to the queue.

As shown in Fig. 7.10, to acquire the lock, a thread sets the `locked` field of its `QNode` to *true*, indicating that the thread is not ready to release the lock. The thread

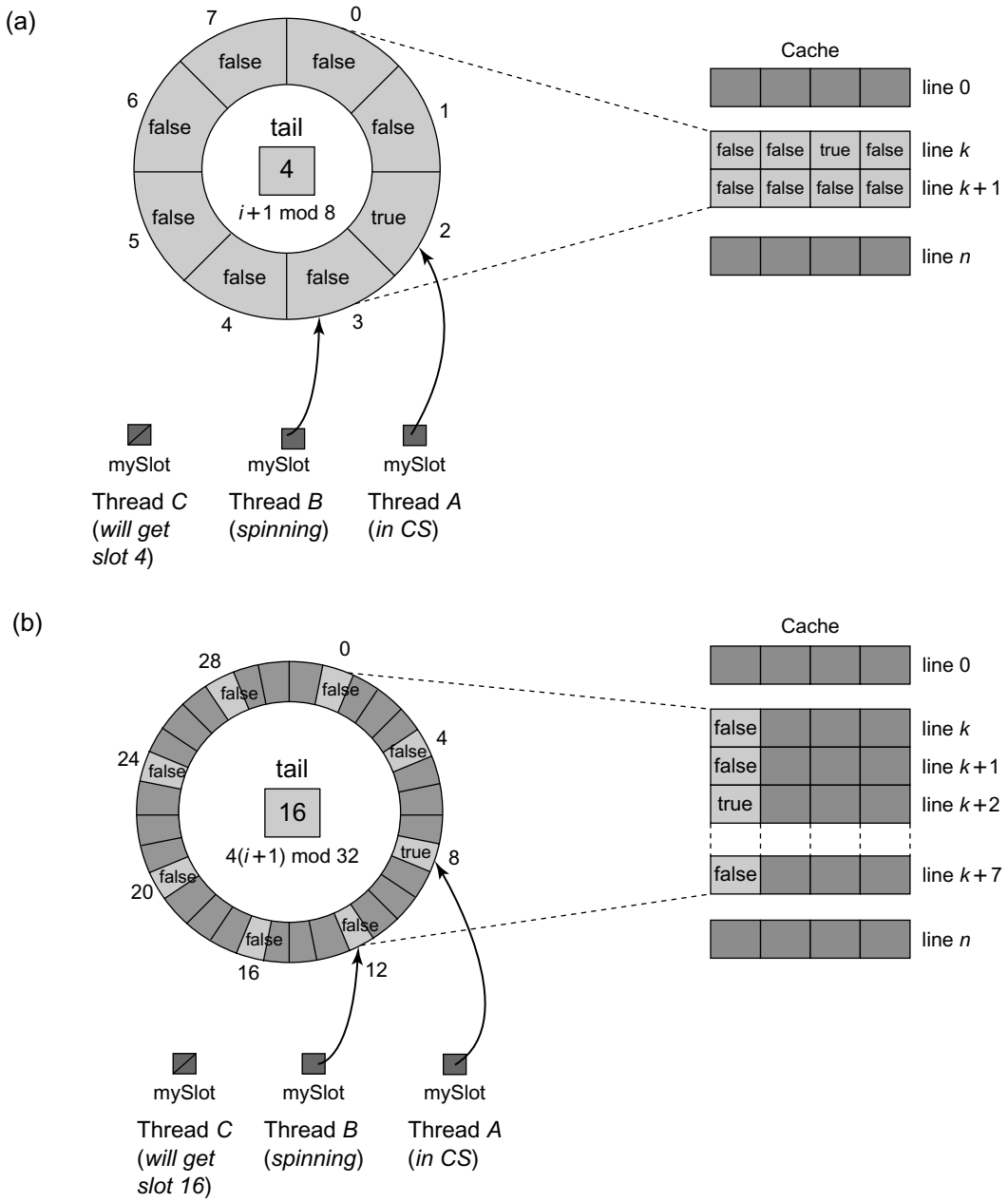

Figure 7.8 The `ALock` with padding to avoid false sharing. In Part (a) the `ALock` has 8 slots which are accessed via a modulo 8 counter. Array entries are typically mapped into cache lines consecutively. As can be seen, when thread *A* changes the status of its entry, thread *B* whose entry is mapped to the same cache line *k* incurs a false invalidation. In Part (b) each location is padded so it is 4 apart from the others with a modulo 32 counter. Even if array entries are mapped consecutively, the entry for *B* is mapped to a different cache line from that of *A*, so if *A* invalidates its entry this does not cause *B* to be invalidated.

```
1  public class CLHLock implements Lock {
2    AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());
3    ThreadLocal<QNode> myPred;
4    ThreadLocal<QNode> myNode;
5    public CLHLock() {
6      tail = new AtomicReference<QNode>(new QNode());
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12     myPred = new ThreadLocal<QNode>() {
13       protected QNode initialValue() {
14         return null;
15       }
16     };
17   }
18   ...
19 }
```

Figure 7.9 The CLHLock class: fields and constructor.

```
20   public void lock() {
21     QNode qnode = myNode.get();
22     qnode.locked = true;
23     QNode pred = tail.getAndSet(qnode);
24     myPred.set(pred);
25     while (pred.locked) {}
26   }
27   public void unlock() {
28     QNode qnode = myNode.get();
29     qnode.locked = false;
30     myNode.set(myPred.get());
31   }
32 }
```

Figure 7.10 The CLHLock class: lock() and unlock() methods.

applies `getAndSet()` to the `tail` field to make its own node the tail of the queue, simultaneously acquiring a reference to its predecessor's `QNode`. The thread then spins on the predecessor's `locked` field until the predecessor releases the lock. To release the lock, the thread sets its node's `locked` field to *false*. It then reuses its predecessor's `QNode` as its new node for future lock accesses. It can do so because at this point the thread's predecessor's `QNode` is no longer used by the predecessor, and the thread's old `QNode` could be referenced both by the thread's successor and by the `tail`.[2] Although we do not do so in our examples, it is possible to recycle

2 It is not necessary for correctness to reuse nodes in garbage-collected languages such as Java or C#, but reuse would be needed in languages such as C++ or C.

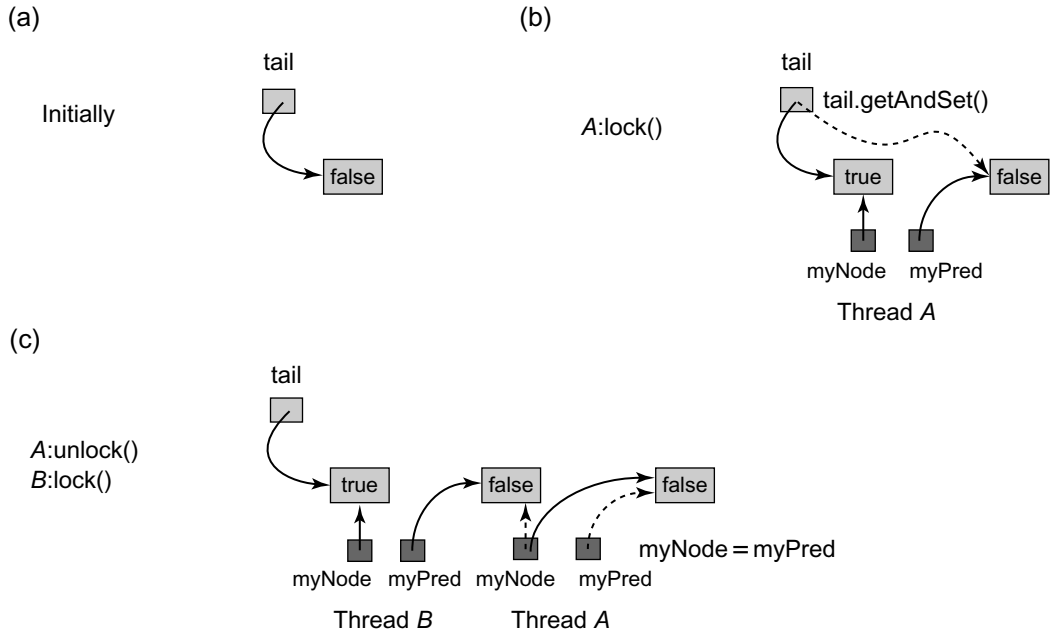

Figure 7.11 `CLHLock` class: lock acquisition and release. Initially the `tail` field refers to a `QNode` whose `locked` field is false. Thread *A* then applies `getAndSet()` to the `tail` field to insert its `QNode` at the tail of the queue, simultaneously acquiring a reference to its predecessor's `QNode`. Next, *B* does the same to insert its `QNode` at the tail of the queue. *A* then releases the lock by setting its node's `locked` field to false. It then recycles the `QNode` referenced by `pred` for future lock accesses.

nodes so that if there are L locks, and each thread accesses at most one lock at a time, then the `CLHLock` class needs only $O(L + n)$ space, as compared to $O(Ln)$ for the `ALock` class. Fig. 7.11 shows a typical `CLHLock` execution.

Like the `ALock`, this algorithm has each thread spin on a distinct location, so when one thread releases its lock, it invalidates only its successor's cache. This algorithm requires much less space than the `ALock` class, and does not require knowledge of the number of threads that might access the lock. Like the `ALock` class, it provides first-come-first-served fairness.

Perhaps the only disadvantage of this lock algorithm is that it performs poorly on cache-less NUMA architectures. Each thread spins waiting for its predecessor's node's `locked` field to become *false*. If this memory location is remote, then performance suffers. On cache-coherent architectures, however, this approach should work well.

7.5.3 The MCS Queue Lock

Fig. 7.12 shows the fields and constructor for the `MCSLock` class. Here, too, the lock is represented as a linked list of `QNode` objects, where each `QNode` represents

```
1  public class MCSLock implements Lock {
2    AtomicReference<QNode> tail;
3    ThreadLocal<QNode> myNode;
4    public MCSLock() {
5      queue = new AtomicReference<QNode>(null);
6      myNode = new ThreadLocal<QNode>() {
7        protected QNode initialValue() {
8          return new QNode();
9        }
10     };
11   }
12   ...
13   class QNode {
14     boolean locked = false;
15     QNode  next = null;
16   }
17 }
```

Figure 7.12 MCSLock class: fields, constructor and QNode class.

```
18   public void lock() {
19     QNode qnode = myNode.get();
20     QNode pred = tail.getAndSet(qnode);
21     if (pred != null) {
22       qnode.locked = true;
23       pred.next = qnode;
24       // wait until predecessor gives up the lock
25       while (qnode.locked) {}
26     }
27   }
28   public void unlock() {
29     QNode qnode = myNode.get();
30     if (qnode.next == null) {
31       if (tail.compareAndSet(qnode, null))
32         return;
33       // wait until predecessor fills in its next field
34       while (qnode.next == null) {}
35     }
36     qnode.next.locked = false;
37     qnode.next = null;
38   }
```

Figure 7.13 MCSLock class: lock() and unlock() methods.

either a lock holder or a thread waiting to acquire the lock. Unlike the CLHLock class, the list is explicit, not virtual: instead of embodying the list in thread-local variables, it is embodied in the (globally accessible) QNode objects, via their next fields.

Fig. 7.13 shows the MCSLock class's lock() and unlock() methods. To acquire the lock, a thread appends its own QNode at the tail of the list (Line 20). If the

queue was not previously empty, it sets the predecessor's `QNode`'s `next` field to refer to its own `QNode`. The thread then spins on a (local) `locked` field in its own `QNode` waiting until its predecessor sets this field to *false* (Lines 21–26).

The `unlock()` method checks whether the node's `next` field is *null* (Line 30). If so, then either no other thread is contending for the lock, or there is another thread, but it is slow. Let q be the thread's current node. To distinguish between these cases, the method applies `compareAndSet`(q, *null*) to the `tail` field. If the call succeeds, then no other thread is trying to acquire the lock, `tail` is set to *null*, and the method returns. Otherwise, another (slow) thread is trying to acquire the lock, so the method spins waiting for it to finish (Line 34). In either case, once the successor has appeared, the `unlock()` method sets its successor's `locked` field to *false*, indicating that the lock is now free. At this point, no other thread can access this `QNode`, and so it can be reused. Fig. 7.14 shows an example execution of the `MCSLock`.

This lock shares the advantages of the `CLHLock`, in particular, the property that each lock release invalidates only the successor's cache entry. It is better suited to cache-less NUMA architectures because each thread controls the location on which it spins. Like the `CLHLock`, nodes can be recycled so that this lock has

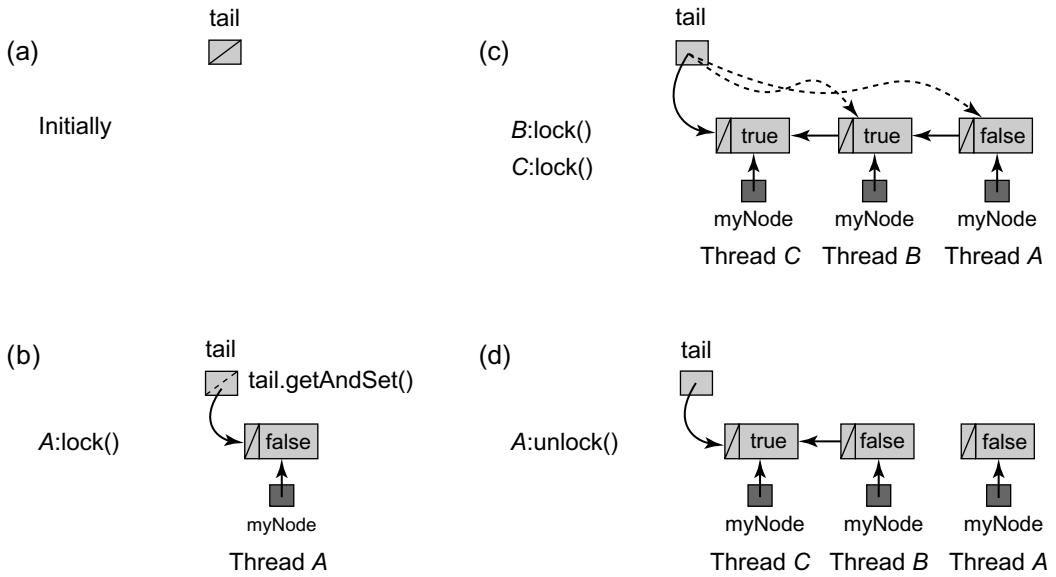

Figure 7.14 A lock acquisition and release in an `MCSLock`. (a) Initially the `tail` is null. (b) To acquire the lock, thread *A* places its own `QNode` at the tail of the list and since it has no predecessor it enters the critical section. (c) thread *B* enqueues its own `QNode` at the tail of the list and modifies its predecessor's `QNode` to refer back to its own. Thread *B* then spins on its `locked` field waiting until *A*, its predecessor, sets this field from true to false. Thread *C* repeats this sequence. (d) To release the lock, *A* follows its `next` field to its successor *B* and sets *B*'s `locked` field to *false*. It can now reuse its `QNode`.

space complexity $O(L + n)$. One drawback of the `MCSLock` algorithm is that releasing a lock requires spinning. Another is that it requires more reads, writes, and `compareAndSet()` calls than the `CLHLock` algorithm.

7.6 A Queue Lock with Timeouts

The Java `Lock` interface includes a `tryLock()` method that allows the caller to specify a *timeout*: a maximum duration the caller is willing to wait to acquire the lock. If the timeout expires before the caller acquires the lock, the attempt is abandoned. A Boolean return value indicates whether the lock attempt succeeded. (For an explanation why these methods throw `InterruptedException`, see Pragma 8.2.3 in Chapter 8.)

Abandoning a `BackoffLock` request is trivial, because a thread can simply return from the `tryLock()` call. Timing out is wait-free, requiring only a constant number of steps. By contrast, timing out any of the queue lock algorithms is far from trivial: if a thread simply returns, the threads queued up behind it will starve.

Here is a bird's-eye view of a queue lock with timeouts. As in the `CLHLock`, the lock is a virtual queue of nodes, and each thread spins on its predecessor's node waiting for the lock to be released. As noted, when a thread times out, it cannot simply abandon its queue node, because its successor will never notice when the lock is released. On the other hand, it seems extremely difficult to unlink a queue node without disrupting concurrent lock releases. Instead, we take a *lazy* approach: when a thread times out, it marks its node as abandoned. Its successor in the queue, if there is one, notices that the node on which it is spinning has been abandoned, and starts spinning on the abandoned node's predecessor. This approach has the added advantage that the successor can recycle the abandoned node.

Fig. 7.15 shows the fields, constructor, and `QNode` class for the `TOLock` (timeout lock) class, a queue lock based on the `CLHLock` class that supports wait-free timeout even for threads in the middle of the list of nodes waiting for the lock.

When a `QNode`'s `pred` field is *null*, the associated thread has either not acquired the lock or has released it. When a `QNode`'s `pred` field refers to a distinguished, static `QNode` called `AVAILABLE`, the associated thread has released the lock. Finally, if the `pred` field refers to some other `QNode`, the associated thread has abandoned the lock request, so the thread owning the successor node should wait on the abandoned node's predecessor.

Fig. 7.16 shows the `TOLock` class's `tryLock()` and `unlock()` methods. The `tryLock()` method creates a new `QNode` with a *null* `pred` field and appends it to the list as in the `CLHLock` class (Lines 5–8). If the lock was free (Line 9), the thread enters the critical section. Otherwise, it spins waiting for its predecessor's `QNode`'s `pred` field to change (Lines 12–19). If the predecessor thread times out, it sets the `pred` field to its own predecessor, and the thread spins instead on the new

```
1  public class TOLock implements Lock{
2    static QNode AVAILABLE = new QNode();
3    AtomicReference<QNode> tail;
4    ThreadLocal<QNode> myNode;
5    public TOLock() {
6      tail = new AtomicReference<QNode>(null);
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12   }
13   ...
14   static class QNode {
15     public QNode pred = null;
16   }
17 }
```

Figure 7.15 `TOLock` class: fields, constructor, and `QNode` class.

```
1    public boolean tryLock(long time, TimeUnit unit)
2        throws InterruptedException {
3      long startTime = System.currentTimeMillis();
4      long patience = TimeUnit.MILLISECONDS.convert(time, unit);
5      QNode qnode = new QNode();
6      myNode.set(qnode);
7      qnode.pred = null;
8      QNode myPred = tail.getAndSet(qnode);
9      if (myPred == null || myPred.pred == AVAILABLE) {
10       return true;
11     }
12     while (System.currentTimeMillis() - startTime < patience) {
13       QNode predPred = myPred.pred;
14       if (predPred == AVAILABLE) {
15         return true;
16       } else if (predPred != null) {
17         myPred = predPred;
18       }
19     }
20     if (!tail.compareAndSet(qnode, myPred))
21       qnode.pred = myPred;
22     return false;
23   }
24   public void unlock() {
25     QNode qnode = myNode.get();
26     if (!tail.compareAndSet(qnode, null))
27       qnode.pred = AVAILABLE;
28   }
29 }
```

Figure 7.16 `TOLock` class: `tryLock()` and `unlock()` methods.

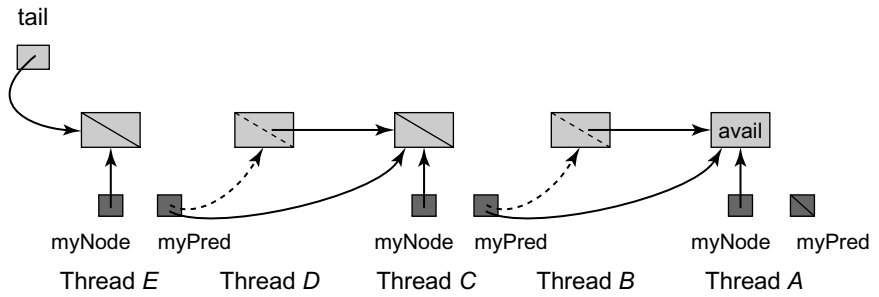

Figure 7.17 Timed-out nodes that must be skipped to acquire the `TOLock`. Threads *B* and *D* have timed out, redirecting their `pred` fields to their predecessors in the list. Thread *C* notices that *B*'s field is directed at *A* and so it starts spinning on *A*. Similarly thread *E* spins waiting for *C*. When *A* completes and sets its `pred` to `AVAILABLE`, *C* will access the critical section and upon leaving it will set its `pred` to `AVAILABLE`, releasing *E*.

predecessor. An example of such a sequence appears in Fig. 7.17. Finally, if the thread itself times out (Line 20), it attempts to remove its `QNode` from the list by applying `compareAndSet()` to the `tail` field. If the `compareAndSet()` call fails, indicating that the thread has a successor, the thread sets its `QNode`'s `pred` field, previously *null*, to its predecessor's `QNode`, indicating that it has abandoned the queue.

In the `unlock()` method, a thread checks, using `compareAndSet()`, whether it has a successor (Line 26), and if so sets its `pred` field to `AVAILABLE`. Note that it is not safe to recycle a thread's old node at this point, since the node may be referenced by its immediate successor, or by a chain of such references. The nodes in such a chain can be recycled as soon as a thread skips over the timed-out nodes and enters the critical section.

The `TOLock` has many of the advantages of the original `CLHLock`: local spinning on a cached location and quick detection that the lock is free. It also has the wait-free timeout property of the `BackoffLock`. However, it has some drawbacks, among them the need to allocate a new node per lock access, and the fact that a thread spinning on the lock may have to go up a chain of timed-out nodes before it can access the critical section.

7.7 A Composite Lock

Spin-lock algorithms impose trade-offs. Queue locks provide first-come-first-served fairness, fast lock release, and low contention, but require nontrivial protocols for recycling abandoned nodes. By contrast, backoff locks support trivial timeout protocols, but are inherently not scalable, and may have slow lock release if timeout parameters are not well-tuned. In this section, we consider an advanced lock algorithm that combines the best of both approaches.

Consider the following simple observation: in a queue lock, only the threads at the front of the queue need to perform lock handoffs. One way to balance the merits of queue locks versus backoff locks is to keep a small number of waiting threads in a queue on the way to the critical section, and have the rest use exponential backoff while attempting to enter this short queue. It is trivial for the threads employing backoff to quit.

The `CompositeLock` class keeps a short, fixed-size array of lock nodes. Each thread that tries to acquire the lock selects a node in the array at random. If that node is in use, the thread backs off (adaptively), and tries again. Once the thread acquires a node, it enqueues that node in a `TOLock`-style queue. The thread spins on the preceding node, and when that node's owner signals it is done, the thread enters the critical section. When it leaves, either because it completed or timed-out, it releases ownership of the node, and another backed-off thread may acquire it. The tricky part of course, is how to recycle the freed nodes of the array while multiple threads attempt to acquire control over them.

The `CompositeLock`'s fields, constructor, and `unlock()` method appear in Fig. 7.18. The `waiting` field is a constant-size `QNode` array, and the `tail` field is an `AtomicStampedReference<QNode>` that combines a reference to the queue tail with a version number needed to avoid the *ABA* problem on updates (see Pragma 10.6.1 of Chapter 10 for a more detailed explanation of

```
1  public class CompositeLock implements Lock{
2    private static final int SIZE = ...;
3    private static final int MIN_BACKOFF = ...;
4    private static final int MAX_BACKOFF = ...;
5    AtomicStampedReference<QNode> tail;
6    QNode[] waiting;
7    Random random;
8    ThreadLocal<QNode> myNode = new ThreadLocal<QNode>() {
9      protected QNode initialValue() { return null; };
10   };
11   public CompositeLock() {
12     tail = new AtomicStampedReference<QNode>(null,0);
13     waiting = new QNode[SIZE];
14     for (int i = 0; i < waiting.length; i++) {
15       waiting[i] = new QNode();
16     }
17     random = new Random();
18   }
19   public void unlock() {
20     QNode acqNode = myNode.get();
21     acqNode.state.set(State.RELEASED);
22     myNode.set(null);
23   }
24   ...
25 }
```

Figure 7.18 The `CompositeLock` class: fields, constructor, and `unlock()` method.

```
1  enum State {FREE, WAITING, RELEASED, ABORTED};
2  class QNode {
3    AtomicReference<State> state;
4    QNode pred;
5    public QNode() {
6      state = new AtomicReference<State>(State.FREE);
7    }
8  }
```

Figure 7.19 The CompositeLock class: the QNode class.

```
1   public boolean tryLock(long time, TimeUnit unit)
2       throws InterruptedException {
3     long patience = TimeUnit.MILLISECONDS.convert(time, unit);
4     long startTime = System.currentTimeMillis();
5     Backoff backoff = new Backoff(MIN_BACKOFF, MAX_BACKOFF);
6     try {
7       QNode node = acquireQNode(backoff, startTime, patience);
8       QNode pred = spliceQNode(node, startTime, patience);
9       waitForPredecessor(pred, node, startTime, patience);
10      return true;
11    } catch (TimeoutException e) {
12      return false;
13    }
14  }
```

Figure 7.20 The CompositeLock class: the tryLock() method.

the AtomicStampedReference<T> class, and Chapter 11 for a more complete discussion of the ABA problem[3]). The tail field is either *null* or refers to the last node inserted in the queue. Fig. 7.19 shows the QNode class. Each QNode includes a State field and a reference to the predecessor node in the queue.

A QNode has four possible states: WAITING, RELEASED, ABORTED, or FREE. A WAITING node is linked into the queue, and the owning thread is either in the critical section, or waiting to enter. A node becomes RELEASED when its owner leaves the critical section and releases the lock. The other two states occur when a thread abandons its attempt to acquire the lock. If the quitting thread has acquired a node but not enqueued it, then it marks the thread as FREE. If the node is enqueued, then it is marked as ABORTED.

Fig. 7.20 shows the tryLock() method. A thread acquires the lock in three steps. First, it *acquires* a node in the waiting array (Line 7), then it enqueues that node in the queue (Line 12), and finally it waits until that node is at the head of the queue (Line 9).

3 ABA is typically a problem only when using dynamically allocated memory in non garbage collected languages. We encounter it here because we are implementing a dynamic linked list using an array.

```
1   private QNode acquireQNode(Backoff backoff, long startTime,
2                              long patience)
3    throws TimeoutException, InterruptedException {
4    QNode node = waiting[random.nextInt(SIZE)];
5    QNode currTail;
6    int[] currStamp = {0};
7    while (true) {
8      if (node.state.compareAndSet(State.FREE, State.WAITING)) {
9        return node;
10     }
11     currTail = tail.get(currStamp);
12     State state = node.state.get();
13     if (state == State.ABORTED || state == State.RELEASED) {
14       if (node == currTail) {
15         QNode myPred = null;
16         if (state == State.ABORTED) {
17           myPred = node.pred;
18         }
19         if (tail.compareAndSet(currTail, myPred,
20             currStamp[0], currStamp[0]+1)) {
21           node.state.set(State.WAITING);
22           return node;
23         }
24       }
25     }
26     backoff.backoff();
27     if (timeout(patience, startTime)) {
28       throw new TimeoutException();
29     }
30   }
31 }
```

Figure 7.21 The `CompositeLock` class: the `acquireQNode()` method.

The algorithm for acquiring a node in the `waiting` array appears in Fig. 7.21. The thread selects a node at random and tries to acquire the node by changing that node's state from `FREE` to `WAITING` (Line 8). If it fails, it examines the node's status. If the node is `ABORTED` or `RELEASED` (Line 13), the thread may "clean up" the node. To avoid synchronization conflicts with other threads, a node can be cleaned up only if it is the last queue node (that is, the value of `tail`). If the tail node is `ABORTED`, `tail` is redirected to that node's predecessor; otherwise `tail` is set to *null*. If, instead, the allocated node is `WAITING`, then the thread backs off and retries. If the thread times out before acquiring its node, it throws `TimeoutException` (Line 28).

Once the thread acquires a node, the `spliceQNode()` method, shown in Fig. 7.22, splices that node into the queue. The thread repeatedly tries to set `tail` to the allocated node. If it times out, it marks the allocated node as `FREE` and throws `TimeoutException`. If it succeeds, it returns the prior value of `tail`, acquired by the node's predecessor in the queue.

```
1   private QNode spliceQNode(QNode node, long startTime, long patience)
2    throws TimeoutException {
3     QNode currTail;
4     int[] currStamp = {0};
5     do {
6       currTail = tail.get(currStamp);
7       if (timeout(startTime, patience)) {
8         node.state.set(State.FREE);
9         throw new TimeoutException();
10      }
11    } while (!tail.compareAndSet(currTail, node,
12        currStamp[0], currStamp[0]+1));
13    return currTail;
14  }
```

Figure 7.22 The CompositeLock class: the spliceQNode() method.

```
1   private void waitForPredecessor(QNode pred, QNode node, long startTime,
2                                   long patience)
3    throws TimeoutException {
4     int[] stamp = {0};
5     if (pred == null) {
6       myNode.set(node);
7       return;
8     }
9     State predState = pred.state.get();
10    while (predState != State.RELEASED) {
11      if (predState == State.ABORTED) {
12        QNode temp = pred;
13        pred = pred.pred;
14        temp.state.set(State.FREE);
15      }
16      if (timeout(patience, startTime)) {
17        node.pred = pred;
18        node.state.set(State.ABORTED);
19        throw new TimeoutException();
20      }
21      predState = pred.state.get();
22    }
23    pred.state.set(State.FREE);
24    myNode.set(node);
25    return;
26  }
```

Figure 7.23 The CompositeLock class: the waitForPredecessor() method.

Finally, once the node has been enqueued, the thread must wait its turn by calling waitForPredecessor() (Fig. 7.23). If the predecessor is *null*, then the thread's node is first in the queue, so the thread saves the node in the thread-local myNode field (for later use by unlock()), and enters the critical section. If the predecessor node is not RELEASED, the thread checks whether it is ABORTED (Line 11).

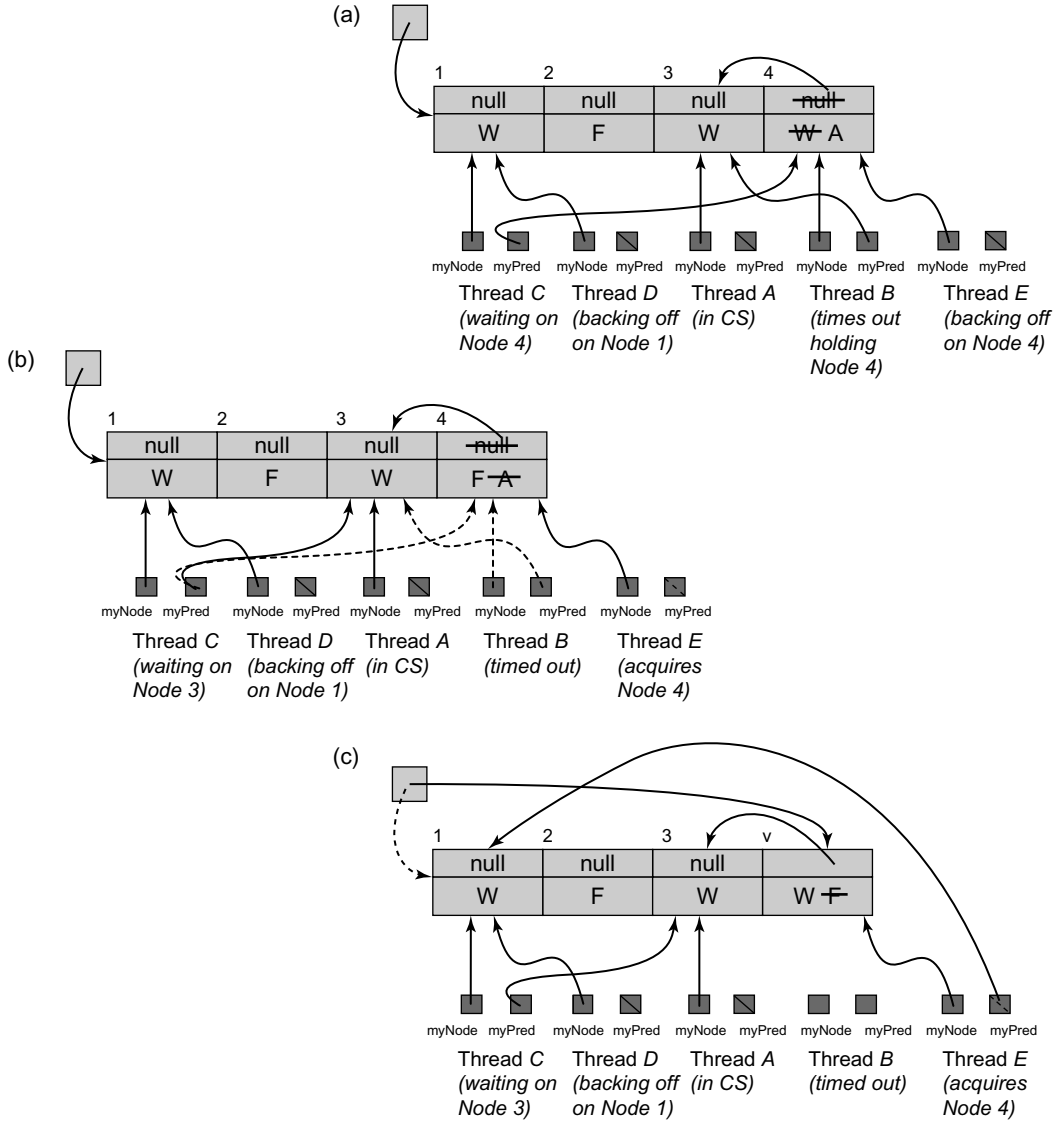

Figure 7.24 The `CompositeLock` class: an execution. In Part (a) thread *A* (which acquired Node 3) is in the critical section. Thread *B* (Node 4) is waiting for *A* to release the critical section and thread *C* (Node 1) is in turn waiting for *B*. Threads *D* and *E* are backing off, waiting to acquire a node. Node 2 is free. The `tail` field refers to Node 1, the last node to be inserted into the queue. At this point *B* times out, inserting an explicit reference to its predecessor, and changing Node 4's state from `WAITING` (denoted by *W*), to `ABORTED` (denoted by *A*). In Part (b), thread *C* cleans up the `ABORTED` Node 4, setting its state to `FREE` and following the explicit reference from 4 to 3 (by redirecting its local `myPred` field). It then starts waiting for *A* (Node 3) to leave the critical section. In Part (c) *E* acquires the `FREE` Node 4, using `compareAndSet()` to set its state to `WAITING`. Thread *E* then inserts Node 4 into the queue, using `compareAndSet()` to swap Node 4 into the tail, then waiting on Node 1, which was previously referred to the tail.

If so, the thread marks the node FREE and waits on the aborted node's predecessor. If the thread times out, then it marks its own node as ABORTED and throws TimeoutException. Otherwise, when the predecessor node becomes RELEASED the thread marks it FREE, records its own node in the thread-local myPred field, and enters the critical section.

The unlock() method (Fig. 7.18) simply retrieves its node from myPred and marks it RELEASED.

The CompositeLock has a number of interesting properties. When threads back off, they access different locations, reducing contention. Lock hand-off is fast, just as in the CLHLock and TOLock algorithms. Abandoning a lock request is trivial for threads in the backoff stage, and relatively straightforward for threads that have acquired queue nodes. For L locks and n threads, the CompositeLock class, requires only $O(L)$ space in the worst case, as compared to the TOLock class's $O(L \cdot n)$. There is one drawback: the CompositeLock class does not guarantee first-come-first-served access.

7.7.1 A Fast-Path Composite Lock

Although the CompositeLock is designed to perform well under contention, performance in the absence of concurrency is also important. Ideally, for a thread running alone, acquiring a lock should be as simple as acquiring an uncontended TASLock. Unfortunately, in the CompositeLock algorithm, a thread running alone must redirect the tail field away from a released node, claim the node, and then splice it into the queue.

A *fast path* is a shortcut through a complex algorithm taken by a thread running alone. We can extend the CompositeLock algorithm to encompass a fast path in which a solitary thread acquires an idle lock without acquiring a node and splicing it into the queue.

Here is a bird's-eye view. We add an extra state, distinguishing between a lock held by an ordinary thread and a lock held by a fast-path thread. If a thread discovers the lock is free, it tries a fast-path acquire. If it succeeds, then it has acquired the lock in a single atomic step. If it fails, then it enqueues itself just as before.

We now examine the algorithm in detail. To reduce code duplication, we define the CompositeFastPathLock class to be a subclass of CompositeLock (see Fig. 7.25).

We use a FASTPATH flag to indicate that a thread has acquired the lock through the fast path. Because we need to manipulate this flag together with the tail field's reference, we "steal" a high-order bit from the tail field's integer stamp (Line 2). The private fastPathLock() method checks whether the tail field's stamp has a clear FASTPATH flag and a *null* reference. If so, it tries to acquire the lock simply by applying compareAndSet() to set the FASTPATH flag to *true*, ensuring that the reference remains *null*. An uncontended lock acquisition thus requires a single atomic operation. The fastPathLock() method returns *true* if it succeeds, and *false* otherwise.

```
1  public class CompositeFastPathLock extends CompositeLock {
2    private static final int FASTPATH = ...;
3    private boolean fastPathLock() {
4      int oldStamp, newStamp;
5      int stamp[] = {0};
6      QNode qnode;
7      qnode = tail.get(stamp);
8      oldStamp = stamp[0];
9      if (qnode != null) {
10       return false;
11     }
12     if ((oldStamp & FASTPATH) != 0) {
13       return false;
14     }
15     newStamp = (oldStamp + 1) | FASTPATH;
16     return tail.compareAndSet(qnode, null, oldStamp, newStamp);
17   }
18   public boolean tryLock(long time, TimeUnit unit)
19     throws InterruptedException {
20     if (fastPathLock()) {
21       return true;
22     }
23     if (super.tryLock(time, unit)) {
24       while ((tail.getStamp() & FASTPATH ) != 0){};
25       return true;
26     }
27     return false;
28   }
```

Figure 7.25 `CompositeFastPathLock` class: the private `fastPathLock()` method returns true if it succeeds in acquiring the lock through the fast path.

```
1    private boolean fastPathUnlock() {
2      int oldStamp, newStamp;
3      oldStamp = tail.getStamp();
4      if ((oldStamp & FASTPATH) == 0) {
5        return false;
6      }    int[] stamp = {0};
7      QNode qnode;
8      do {
9        qnode = tail.get(stamp);
10       oldStamp = stamp[0];
11       newStamp = oldStamp & (^FASTPATH);
12     } while (!tail.compareAndSet(qnode, qnode, oldStamp, newStamp));
13     return true;
14   }
15   public void unlock() {
16     if (!fastPathUnlock()) {
17       super.unlock();
18     };
19   }
```

Figure 7.26 `CompositeFastPathLock` class: `fastPathLock()` and `unlock()` methods.

The tryLock() method (Lines 18–28) first tries the fast path by calling fastPathLock(). If it fails, then it pursues the slow path by calling the CompositeLock class's tryLock() method. Before it can return from the slow path, however, it must ensure that no other thread holds the fast-path lock by waiting until the FASTPATH flag is clear (Line 24).

The fastPathUnlock() method returns *false* if the fast-path flag is not set (Line 4). Otherwise, it repeatedly tries to clear the flag, leaving the reference component unchanged (Lines 8–12), returning *true* when it succeeds.

The CompositeFastPathLock class's unlock() method first calls fastPathUnlock() (Line 16). If that call fails to release the lock, it then calls the CompositeLock's unlock() method (Line 17).

7.8 Hierarchical Locks

Many of today's cache-coherent architectures organize processors in *clusters*, where communication within a cluster is significantly faster than communication between clusters. For example, a cluster might correspond to a group of processors that share memory through a fast interconnect, or it might correspond to the threads running on a single core in a multicore architecture. We would like to design locks that are sensitive to these differences in locality. Such locks are called *hierarchical* because they take into account the architecture's memory hierarchy and access costs.

Architectures can easily have two, three, or more levels of memory hierarchy, but to keep things simple, we assume there are two. We consider an architecture consisting of clusters of processors, where processors in the same cluster communicate efficiently through a shared cache. Inter-cluster communication is significantly more expensive than intra-cluster communication.

We assume that each cluster has a unique *cluster id* known to each thread in the cluster, available via ThreadID.getCluster(). Threads do not migrate between clusters.

7.8.1 A Hierarchical Backoff Lock

A test–and–test–and–set lock can easily be adapted to exploit clustering. Suppose the lock is held by thread *A*. If threads from *A*'s cluster have shorter backoff times, then when the lock is released, local threads are more likely to acquire the lock than remote threads, reducing the overall time needed to switch lock ownership. Fig. 7.27 shows the HBOLock class, a hierarchical backoff lock based on this principle.

One drawback of the HBOLock is that it may be *too* successful in exploiting locality. There is a danger that threads from the same cluster will repeatedly transfer the lock among themselves while threads from other clusters starve. Moreover, acquiring and releasing the lock invalidates remotely cached copies of the lock field, which can be expensive on cache-coherent NUMA architectures.

```
1  public class HBOLock implements Lock {
2    private static final int LOCAL_MIN_DELAY = ...;
3    private static final int LOCAL_MAX_DELAY = ...;
4    private static final int REMOTE_MIN_DELAY = ...;
5    private static final int REMOTE_MAX_DELAY = ...;
6    private static final int FREE = -1;
7    AtomicInteger state;
8    public HBOLock() {
9      state = new AtomicInteger(FREE);
10   }
11   public void lock() {
12     int myCluster = ThreadID.getCluster();
13     Backoff localBackoff =
14         new Backoff(LOCAL_MIN_DELAY, LOCAL_MAX_DELAY);
15     Backoff remoteBackoff =
16         new Backoff(REMOTE_MIN_DELAY, REMOTE_MAX_DELAY);
17     while (true) {
18       if (state.compareAndSet(FREE, myCluster)) {
19         return;
20       }
21       int lockState = state.get();
22       if (lockState == myCluster) {
23         localBackoff.backoff();
24       } else {
25         remoteBackoff.backoff();
26       }
27     }
28   }
29   public void unlock() {
30     state.set(FREE);
31   }
32 }
```

Figure 7.27 The HBOLock class: a hierarchical backoff lock.

7.8.2 A Hierarchical CLH Queue Lock

To provide a more balanced way to exploit clustering, we now consider the design of a hierarchical queue lock. The challenge is to reconcile conflicting fairness requirements. We would like to favor transferring locks within the same cluster to avoid high communication costs, but we also want to ensure some degree of fairness, so that remote lock requests are not excessively postponed in favor of local requests. We balance these demands by scheduling *sequences* of requests from the same cluster together.

The HCLHLock queue lock (Fig. 7.28) consists of a collection of *local queues*, one per cluster, and a single *global queue*. Each queue is a linked list of nodes, where the links are implicit, in the sense that they are held in thread-local fields, myQNode and myPred.

We say that a thread *owns* its myQNode node. For any node in a queue (other than at the head), its predecessor is its owner's myPred node. Fig. 7.30

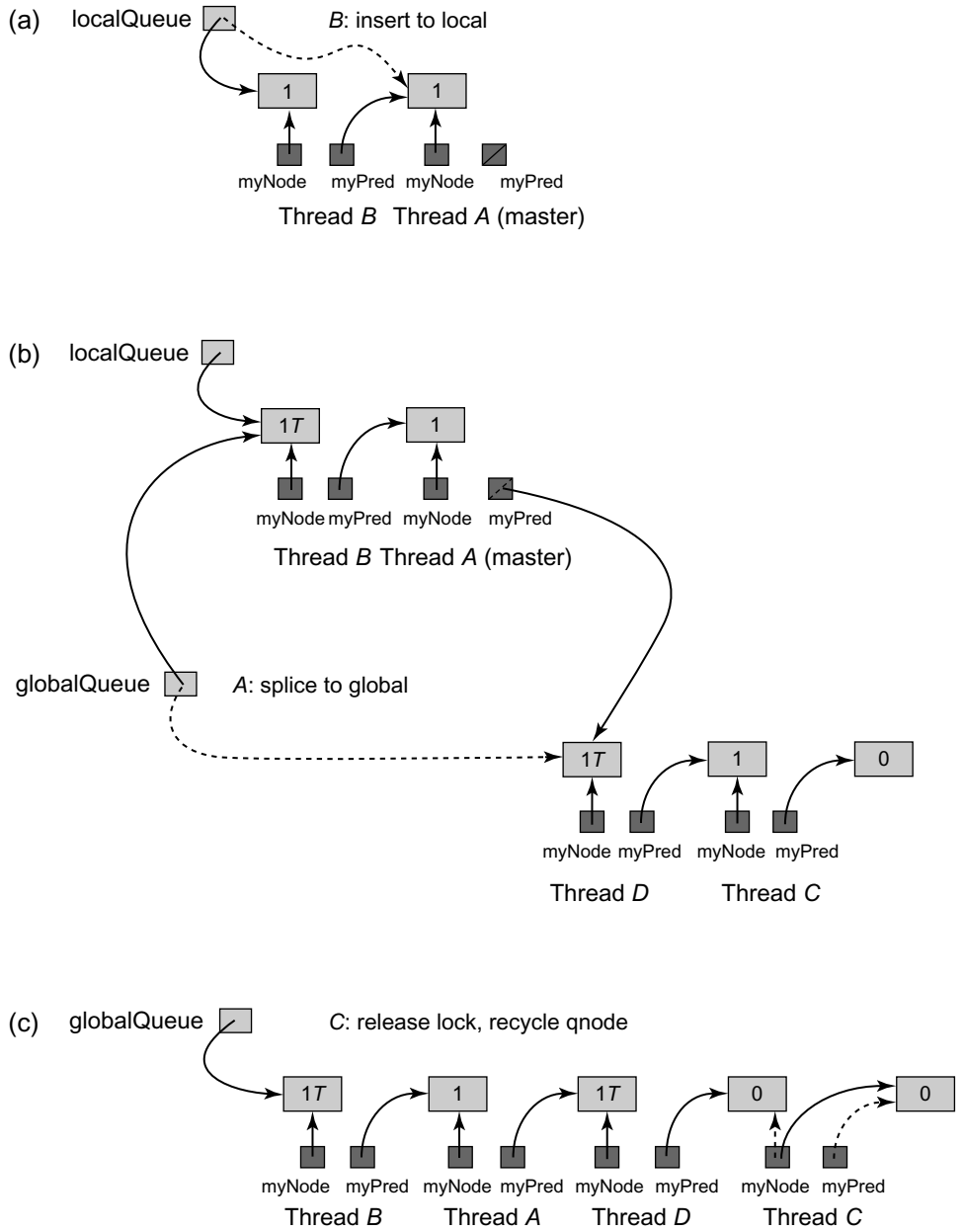

Figure 7.28 Lock acquisition and release in a `HCLHLock`. The `successorMustWait` field is marked in the nodes by a 0 (for false) or a 1 (for true). A node is marked as a local tail when it is being spliced by adding the symbol *T*. In Part (a), *B* inserts its node into the local queue. In Part (b), *A* splices the local queue containing *A* and *B*'s nodes onto the global queue, which already contains *C* and *D*'s nodes. In Part (c), *C* releases the lock by setting its node's `successorMustWait` flag to false, and then setting `myQNode` to the predecessor node.

```
1  public class HCLHLock implements Lock {
2    static final int MAX_CLUSTERS = ...;
3    List<AtomicReference<QNode>> localQueues;
4    AtomicReference<QNode> globalQueue;
5    ThreadLocal<QNode> currNode = new ThreadLocal<QNode>() {
6      protected QNode initialValue() { return new QNode(); };
7    };
8    ThreadLocal<QNode> predNode = new ThreadLocal<QNode>() {
9      protected QNode initialValue() { return null; };
10   };
11   public HCLHLock() {
12     localQueues = new ArrayList<AtomicReference<QNode>>(MAX_CLUSTERS);
13     for (int i = 0; i < MAX_CLUSTERS; i++) {
14       localQueues.add(new AtomicReference <QNode>());
15     }
16     QNode head = new QNode();
17     globalQueue = new AtomicReference<QNode>(head);
18   }
```

Figure 7.29 The HCLHLock class: fields and constructor.

```
1    class QNode {
2      // private boolean tailWhenSpliced;
3      private static final int TWS_MASK = 0x80000000;
4      // private boolean successorMustWait = false;
5      private static final int SMW_MASK = 0x40000000;
6      // private int clusterID;
7      private static final int CLUSTER_MASK = 0x3FFFFFFF;
8      AtomicInteger state;
9      public QNode() {
10       state = new AtomicInteger(0);
11     }
12     public void unlock() {
13       int oldState = 0;
14       int newState = ThreadID.getCluster();
15       // successorMustWait = true;
16       newState |= SMW_MASK;
17       // tailWhenSpliced = false;
18       newState &= (^TWS_MASK);
19       do {
20         oldState = state.get();
21       } while (! state.compareAndSet(oldState, newState));
22     }
23     public int getClusterID() {
24       return state.get() & CLUSTER_MASK;
25     }
26     // other getters and setters omitted.
27   }
```

Figure 7.30 The HCLHLock class: the inner QNode class.

shows the QNode class. Each node has three virtual fields: the current (or most recent) owner's ClusterId, and two Boolean fields, successorMustWait and tailWhenSpliced. These fields are virtual in the sense that they need to be updated atomically, so we represent them as bit-fields in an AtomicInteger field, using simple masking and shifting operations to extract their values. The tailWhenSpliced field indicates whether the node is the last node in the sequence currently being spliced onto the global queue. The successorMustWait field is the same as in the original CLH algorithm: it is set to *true* before being enqueued, and set to *false* by the node's owner on releasing the lock. Thus, a thread waiting to acquire the lock may proceed when its predecessor's successorMustWait field becomes *false*. Because we need to update these fields atomically, they are private, accessed indirectly through synchronized methods.

Fig. 7.28 illustrates how the HCLHLock class acquires and releases a lock. The lock() method first adds the thread's node to the local queue, and then waits until either the thread can enter the critical section or its node is at the head of the local queue. In the latter case, we say the thread is the *cluster master*, and it is responsible for splicing the local queue onto the global queue.

The code for the lock() method appears in Fig. 7.31. The thread's node has been initialized so that successorMustWait is *true*, tailWhenSpliced is *false*, and the ClusterId field is the caller's cluster. The thread then adds its node to the end (tail) of its local cluster's queue, using compareAndSet() to change the tail to its node (Line 9). Upon success, the thread sets its myPred to the node it replaced as the tail. We call this node the *predecessor*.

The thread then calls waitForGrantOrClusterMaster() (Line 11), which causes the thread to spin until one of the following conditions is true:

1. the predecessor node is from the same cluster, and tailWhenSpliced and successorMustWait are both *false*, or
2. the predecessor node is from a different cluster or the predecessor's flag tailWhenSpliced is *true*.

In the first case, the thread's node is at the head of the global queue, so it enters the critical section and returns (Line 14). In the second case, as explained here, the thread's node is at the head of the local queue, so the thread is the cluster master, making it responsible for splicing the local queue onto the global queue. (If there is no predecessor, that is, if the local queue's tail is *null*, then the thread becomes the cluster master immediately.) Most of the spinning required by waitForGrantOrClusterMaster() is local and incurs little or no communication cost.

Otherwise, either the predecessor's cluster is different from the thread's, or the predecessor's tailWhenSpliced flag is *true*. If the predecessor belongs to a different cluster, then it cannot be in this thread's local queue. The predecessor must have already been moved to the global queue and recycled to a thread in a different cluster. On the other hand, if the predecessor's tailWhenSpliced flag

```
1   public void lock() {
2     QNode myNode = currNode.get();
3     AtomicReference<QNode> localQueue = localQueues.get
4                                         (ThreadID.getCluster());
5     // splice my QNode into local queue
6     QNode myPred = null;
7     do {
8       myPred = localQueue.get();
9     } while (!localQueue.compareAndSet(myPred, myNode));
10    if (myPred != null) {
11      boolean iOwnLock = myPred.waitForGrantOrClusterMaster();
12      if (iOwnLock) {
13        predNode.set(myPred);
14        return;
15      }
16    }
17    // I am the cluster master: splice local queue into global queue.
18    QNode localTail = null;
19    do {
20      myPred = globalQueue.get();
21      localTail = localQueue.get();
22    } while(!globalQueue.compareAndSet(myPred, localTail));
23    // inform successor it is the new master
24    localTail.setTailWhenSpliced(true);
25    while (myPred.isSuccessorMustWait()) {};
26    predNode.set(myPred);
27    return;
28  }
```

Figure 7.31 The `HCLHLock` class: `lock()` method. As in the `CLHLock`, `lock()` saves the predecessor's recently released node to be used for next lock acquisition attempt.

is *true*, then the predecessor node was the last that moved to the global queue, and therefore the thread's node is now at the head of the local queue. It cannot have been moved to the global queue because only the cluster master, the thread whose node is at the head of the local queue, moves nodes onto the global queue.

As cluster master, a thread's role is to splice the nodes accumulated in the local queue onto the global queue. The threads in the local queue spin, each on its predecessor's node. The cluster master reads the local queue's tail and calls `compareAndSet()` to change the global queue's tail to the node it saw at the tail of its local queue (Line 22). When it succeeds, `myPred` is the tail of the global queue that it replaced (Line 20). It then sets to *true* the `tailWhenSpliced` flag of the last node it spliced onto the global queue (Line 24), indicating to that node's (local) successor that it is now the head of the local queue. This sequence of operations transfers the local nodes (up to the local tail) into the CLH-style global queue in the same order as in the local queue.

```
29    public void unlock() {
30      QNode myNode = currNode.get();
31      myNode.setSuccessorMustWait(false);
32      QNode node = predNode.get();
33      node.unlock();
34      currNode.set(node);
35    }
```

Figure 7.32 The HCLHLock class: unlock() method. This method promotes the node saved by the lock() operation and initializes the QNode to be used in the next lock acquisition attempt.

Once in the global queue, the cluster master acts as though it were in an ordinary CLHLock queue, entering the critical section when its (new) predecessor's successorMustWait field is *false* (Line 25). The other threads whose nodes were spliced in are not aware that anything has changed, so they continue spinning as before. Each will enter the critical section when its predecessor's successorMustWait field becomes *false*.

As in the original CLHLock algorithm, a thread releases the lock by setting its node's successorMustWait field to *false* (Fig. 7.32). When unlocking, the thread saves its predecessor's node to be used in its next lock acquisition attempt (Line 34).

The HCLHLock lock favors sequences of local threads, one waiting for the other, within the waiting list in the global queue. As with the CLHLock lock, the use of implicit references minimizes cache misses, and threads spin on locally cached copies of their successor's node state.

7.9 One Lock To Rule Them All

In this chapter, we have seen a variety of spin locks that vary in characteristics and performance. Such a variety is useful, because no single algorithm is ideal for all applications. For some applications, complex algorithms work best, and for others, simple algorithms are preferable. The best choice usually depends on specific aspects of the application and the target architecture.

7.10 Chapter Notes

The TTASLock is due to Clyde Kruskal, Larry Rudolph, and Marc Snir [87]. Exponential back off is a well-known technique used in Ethernet routing, presented in the context of multiprocessor mutual exclusion by Anant Agarwal and

Mathews Cherian [6]. Tom Anderson [14] invented the `ALock` algorithm and was one of the first to empirically study the performance of spin locks in shared memory multiprocessors. The `MCSLock`, due to John Mellor-Crummey and Michael Scott [114], is perhaps the best-known queue lock algorithm. Today's Java Virtual Machines use object synchronization based on simplified monitor algorithms such as the *Thinlock* of David Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano [17], the *Metalock* of Ole Agesen, Dave Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna and Derek White [7], or the *RelaxedLock* of Dave Dice [31]. All these algorithms are variations of the `MCSLock` lock.

The `CLHLock` lock is due to Travis Craig, Erik Hagersten, and Anders Landin [30, 111]. The `TOLock` with nonblocking timeout is due to Bill Scherer and Michael Scott [138, 139]. The `CompositeLock` and its variations are due to Virendra Marathe, Mark Moir, and Nir Shavit [121]. The notion of using a fast-path in a mutual exclusion algorithm is due to Leslie Lamport [96]. Hierarchical locks were invented by Zoran Radović and Erik Hagersten. The `HBOLock` is a variant of their original algorithm [131] and the particular `HCLHLock` presented here is due to Victor Luchangco, Daniel Nussbaum, and Nir Shavit [110].

Danny Hendler, Faith Fich, and Nir Shavit [39] have extended the work of Jim Burns and Nancy Lynch to show that any starvation-free mutual exclusion algorithm requires $\Omega(n)$ space, even if strong operations such as `getAndSet()` or `compareAndSet()` are used, implying that all the queue-lock algorithms considered here are space-optimal.

The schematic performance graph in this chapter is loosely based on empirical studies by Tom Anderson [14], as well as on data collected by the authors on various modern machines. We chose to use schematics rather than actual data because of the great variation in machine architectures and their significant effect on lock performance.

The Sherlock Holmes quote is from *The Sign of Four* [36].

7.11 Exercises

Exercise 85. Fig. 7.33 shows an alternative implementation of `CLHLock` in which a thread reuses its own node instead of its predecessor node. Explain how this implementation can go wrong.

Exercise 86. Imagine n threads, each of which executes method `foo()` followed by method `bar()`. Suppose we want to make sure that no thread starts `bar()` until all threads have finished `foo()`. For this kind of synchronization, we place a *barrier* between `foo()` and `bar()`.

First barrier implementation: We have a counter protected by a test–and–test–and–set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n.

```
1  public class BadCLHLock implements Lock {
2    // most recent lock holder
3    AtomicReference<Qnode> tail;
4    // thread-local variable
5    ThreadLocal<Qnode> myNode;
6    public void lock() {
7      Qnode qnode = myNode.get();
8      qnode.locked = true;     // I'm not done
9      // Make me the new tail, and find my predecessor
10     Qnode pred = tail.getAndSet(qnode);
11     // spin while predecessor holds lock
12     while (pred.locked) {}
13   }
14   public void unlock() {
15     // reuse my node next time
16     myNode.get().locked = false;
17   }
18   static class Qnode { // Queue node inner class
19     public boolean locked = false;
20   }
21 }
```

Figure 7.33 An incorrect attempt to implement a CLHLock.

Second barrier implementation: We have an n-element Boolean array b, all *false*. Thread zero sets b[0] to *true*. Every thread i, for $0 < i \leqslant n$, spins until b[$i-1$] is *true*, sets b[i] to *true*, and proceeds.

Compare the behavior of these two implementations on a bus-based cache-coherent architecture.

Exercise 87. Prove that the CompositeFastPathLock implementation guarantees mutual exclusion, but is not starvation-free.

Exercise 88. Notice that, in the HCLHLock lock, for a given cluster master thread, in the interval between setting the global tail reference and raising the tailWhenSpliced flag of the last spliced node, the nodes spliced onto the global queue are in both its local queue and the global queue. Explain why the algorithm is still correct.

Exercise 89. Notice that, in the HCLHLock lock, what will happen if the time between becoming cluster master and successfully splicing the local queue into the global queue is too small? Suggest a remedy to this problem.

Exercise 90. Why is it important that the fields of the State object accessed by the HCLHLock lock's waitForGrantOrClusterMaster() method be read and modified atomically? Provide the code for the HCLHLock lock's waitForGrantOrClusterMaster() method. Does your implementation require the use of a compareAndSet(), and if so, can it be implemented efficiently without it?

Exercise 91. Design an `isLocked`() method that tests whether a thread is holding a lock (but does not acquire that lock). Give implementations for

- Any `testAndSet`() spin lock
- The CLH queue lock, and
- The MCS queue lock.

Exercise 92. (Hard) Where does the $\Omega(n)$ space complexity lower bound proof for deadlock-free mutual exclusion of Chapter 2 break when locks are allowed to use read–modify–write operations?