18 Transactional Memory

18.1 Introduction

We now turn our attention from devising data structures and algorithms to critiquing the tools we use to solve these problems. These tools are the synchronization primitives provided by today's architectures, encompassing various kinds of locking, both spinning and blocking, and atomic operations such as `compareAndSet()` and its relatives. They have mostly served us well. We, the community of multiprocessor programmers, have been able to construct many useful and elegant data structures. Nevertheless, everyone knows that *the tools are flawed.* In this chapter, we review and analyze the strengths and weaknesses of the standard synchronization primitives, and describe some emerging alternatives that are likely to extend, and perhaps even to displace many of today's standard primitives.

18.1.1 What is Wrong with Locking?

Locking, as a synchronization discipline, has many pitfalls for the inexperienced programmer. *Priority inversion* occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. *Convoying* occurs when a thread holding a lock is descheduled, perhaps by exhausting its scheduling quantum by a page fault, or by some other kind of interrupt. While the thread holding the lock is inactive, other threads that require that lock will queue up, unable to progress. Even after the lock is released, it may take some time to drain the queue, in much the same way that an accident can slow traffic even after the debris has been cleared away. *Deadlock* can occur if threads attempt to lock the same objects in different orders. Deadlock avoidance can be awkward if threads must lock many objects, particularly if the set of objects is not known in advance. In the past, when highly scalable applications were rare and valuable,

```
/*
 * When a locked buffer is visible to the I/O layer BH_Launder
 * is set. This means before unlocking we must clear BH_Launder,
 * mb() on alpha and then clear BH_Lock, so no reader can see
 * BH_Launder set on an unlocked buffer and then risk to deadlock.
 */
```

Figure 18.1 Synchronization by convention: a typical comment from the Linux kernel.

these hazards were avoided by deploying teams of dedicated expert programmers. Today, when highly scalable applications are becoming commonplace, the conventional approach is just too expensive.

The heart of the problem is that no one really knows how to organize and maintain large systems that rely on locking. The association between locks and data is established mostly by convention. Ultimately, it exists only in the mind of the programmer, and may be documented only in comments. Fig. 18.1 shows a typical comment from a Linux header file[1] describing the conventions governing the use of a particular kind of buffer. Over time, interpreting and observing many such conventions spelled out in this way may complicate code maintenance.

18.1.2 What is Wrong with `compareAndSet()`?

One way to bypass the problems of locking is to rely on atomic primitives like `compareAndSet()`. Algorithms that use `compareAndSet()` and its relatives are often hard to devise, and sometimes, though not always, have a high overhead. The principal difficulty is that nearly all synchronization primitives, whether reading, writing, or applying an atomic `compareAndSet()`, operate only on a single word. This restriction often forces a complex and unnatural structure on algorithms.

Let us review the lock-free queue of Chapter 10 (reproduced in Fig. 18.2), this time with an eye toward the underlying synchronization primitives.

A complication arises between Lines 12 and 13. The `enq()` method calls `compareAndSet()` to change both the `tail` node's `next` field and the `tail` field itself to the new node. Ideally, we would like to atomically combine both `compareAndSet()` calls, but because these calls occur one-at-a-time both `enq()` and `deq()` must be prepared to encounter a half-finished `enq()` (Line 12). One way to address this problem is to introduce a `multiCompareAndSet()` primitive, as shown in Fig. 18.3. This method takes as arguments an array of `AtomicReference<T>` objects, an array of expected `T` values, and an array of `T`-values used for updates. It performs a simultaneous `compareAndSet()` on all

1 Kernel v2.4.19 `/fs/buffer.c`


```
1  public class LockFreeQueue<T> {
2    private AtomicReference<Node> head;
3    private AtomicReference<Node> tail;
4    ...
5    public void enq(T item) {
6      Node node = new Node(item);
7      while (true) {
8        Node last = tail.get();
9        Node next = last.next.get();
10       if (last == tail.get()) {
11         if (next == null) {
12           if (last.next.compareAndSet(next, node)) {
13             tail.compareAndSet(last, node);
14             return;
15           }
16         } else {
17           tail.compareAndSet(last, next);
18         }
19       }
20     }
21   }
22 }
```

Figure 18.2 The `LockFreeQueue` class: the `enq()` method.

```
1  <T> boolean multiCompareAndSet(
2        AtomicReference<T>[] target,
3        T[] expected,
4        T[] update) {
5    atomic {
6      for (int i = 0; i < target.length)
7        if (!target[i].get().equals(expected[i].get()))
8          return false;
9      for (int i = 0; i < target.length)
10       target[i].set(update[i].get);
11     return true;
12   }
13 }
```

Figure 18.3 Pseudocode for `multiCompareAndSet()`. This code is executed atomically.

array elements, and if any one fails, they all do. In more detail: if, for all i, the value of `target`$[i]$ is `expected`$[i]$, then set `target`$[i]$'s value to `update`$[i]$ and return *true*. Otherwise leave `target`$[i]$ unchanged, and return *false*.

Note that there is no obvious way to implement `multiCompareAndSet()` on conventional architectures. If there were, comparing the `LockFreeQueue` implementations in Figs. 18.2 and 18.4 illustrates how `multiCompareAndSet()` simplifies concurrent data structures. The complex logic of Lines 11–12 is replaced by a call to a single `multiCompareAndSet()` call.

```
1   public void enq(T item) {
2     Node node = new Node(item);
3     while (true) {
4       Node last = tail.get();
5       Node next = last.next.get();
6       if (last == tail.get()) {
7         AtomicReference[] target = {last.next, tail};
8         T[] expect = {next, last};
9         T[] update = {node, node};
10        if (multiCompareAndSet(target, expect, update)) return;
11      }
12    }
13  }
```

Figure 18.4 The `LockFreeQueue` class: simplified `enq()` method with `multiCompareAndSet()`.

While multi-word extensions such as `multiCompareAndSet()` might be useful, they do not help with another serious flaw, discussed in Section 18.1.3.

18.1.3 What is Wrong with Compositionality?

All the synchronization mechanisms we have considered so far, with or without locks, have a major drawback: they cannot easily be *composed*. Let us imagine that we want to dequeue an item x from queue `q0` and enqueue it at another, `q1`. The transfer must be *atomic*: no concurrent thread should observe either that x has vanished, or that it is present in both queues. In `Queue` implementations based on monitors, each method acquires the lock internally, so it is essentially impossible to combine two method calls in this way.

Failure to compose is not restricted to mutual exclusion. Let us consider a bounded queue class whose `deq()` method blocks as long as the queue is empty (using either `wait/notify` or explicit condition objects). We imagine that we have two such queues, and we want to dequeue an item from *either* queue. If both queues are empty, then we want to block until an item shows up in either one. In `Queue` implementations based on monitors, each method provides its own conditional waiting, so it is essentially impossible to wait on two conditions in this way.

Naturally, there are always *ad hoc* solutions. For the atomic transfer, we could introduce a lock to be acquired by any thread attempting an atomic modification to both `q0` and `q1`. But such a lock would be a concurrency bottleneck (no concurrent transfers) and it requires knowing in advance the identities of the two queues. Or, the queues themselves might export their synchronization state, (say, via `lock()` and `unlock()` methods), and rely on the caller to manage multi-object synchronization. Exposing synchronization state in this way would have a devastating effect on modularity, complicating interfaces, and relying on callers to follow complicated conventions. Also, this approach simply would not work for nonblocking queue implementations.

18.1.4 What can We Do about It?

We can summarize the problems with conventional synchronization primitives as follows.

- Locks are hard to manage effectively, especially in large systems.
- Atomic primitives such as `compareAndSet()` operate on only one word at a time, resulting in complex algorithms.
- It is difficult to compose multiple calls to multiple objects into atomic units.

In Section 18.2, we introduce *transactional memory*, an emerging programming model that proposes a solution to each of these problems.

18.2 Transactions and Atomicity

A *transaction* is a sequence of steps executed by a single thread. Transactions must be *serializable*, meaning that they appear to execute sequentially, in a one-at-a-time order. Serializability is a kind of coarse-grained version of linearizability. Linearizability defined atomicity of individual objects by requiring that each method call of a given object appear to take effect instantaneously between its invocation and response, Serializability, on the other hand, defines atomicity for entire transactions, that is, blocks of code that may include calls to multiple objects. It ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call.[2] Properly implemented, transactions do not deadlock or livelock.

We now describe some simple programming language extensions to Java that support a transactional model of synchronization. These extensions are not currently part of Java, but they illustrate the model. The features described here are a kind of average of features provided by contemporary transactional memory systems. Not all systems provide all these features: some provide weaker guarantees, some stronger. Nevertheless, understanding these features will go a long way toward understanding modern transactional memory models.

The **`atomic`** keyword delimits a transaction in much the same way the **`synchronized`** keyword delimits a critical section. While **`synchronized`** blocks acquire a specific lock, and are atomic only with respect to other **`synchronized`** blocks that acquire the same lock, an **`atomic`** block is atomic with respect to all

2 Some definitions of serializability in the literature do not require transactions to be serialized in an order compatible with their real-time precedence order.

```
1  public class TransactionalQueue<T> {
2    private Node head;
3    private Node tail;
4    public TransactionalQueue() {
5      Node sentinel = new Node(null);
6      head = sentinel;
7      tail = sentinel;
8    }
9    public void enq(T item) {
10     atomic {
11       Node node = new Node(item);
12       tail.next = node;
13       tail = node;
14     }
15   }
```

Figure 18.5 An unbounded transactional queue: the `enq()` method.

other **atomic** blocks. Nested **synchronized** blocks can deadlock if they acquire locks in opposite orders, while nested **atomic** blocks cannot.

Because transactions allow atomic updates to multiple locations, they eliminate the need for `multiCompareAndSet()`. Fig. 18.5 shows the `enq()` method for a transactional queue. Let us compare this code with the lock-free code of Fig. 18.2: there is no need for `AtomicReference` fields, `compareAndSet()` calls, or retry loops. Here, the code is essentially sequential code bracketed by **atomic** blocks.

To explain how transactions are used to write concurrent programs, it is convenient to say something about how they are implemented. Transactions are executed *speculatively*: as a transaction executes, it makes *tentative* changes to objects. If it completes without encountering a synchronization conflict, then it *commits* (the tentative changes become permanent) or it *aborts* (the tentative changes are discarded).

Transactions can be nested. Transactions must be nested for simple modularity: one method should be able to start a transaction and then call another method without caring whether the nested call starts a transaction. Nested transactions are especially useful if a nested transaction can abort without aborting its parent. This property will be important when we discuss conditional synchronization later on.

Recall that atomically transferring an item from one queue to another was essentially impossible with objects that use internal monitor locks. With transactions, composing such atomic method calls is almost trivial. Fig. 18.7 shows how to compose a `deq()` call that dequeues an item x from a queue `q0` and an `enq`(x) call that enqueues that item to another queue `q1`.

What about conditional synchronization? Fig. 18.6 shows the `enq()` method for a bounded buffer. The method enters an **atomic** block (Line 2), and tests whether the buffer is full (Line 3). If so, it calls `retry` (Line 4), which rolls


```
1   public void enq(T x) {
2     atomic {
3       if (count == items.length)
4         retry;
5       items[tail] = x;
6       if (++tail == items.length)
7         tail = 0;
8       ++count;
9     }
10  }
```

Figure 18.6 A bounded transactional queue: the `enq()` method with `retry`.

```
1   atomic {
2     x = q0.deq();
3     q1.enq(x);
4   }
```

Figure 18.7 Composing atomic method calls.

```
1   atomic {
2     x = q0.deq();
3   } orElse {
4     x = q1.deq();
5   }
```

Figure 18.8 The `orElse` statement: waiting on multiple conditions.

back the enclosing transaction, pauses it, and restarts it when the object state has changed. Conditional synchronization is one reason it may be convenient to roll back a nested transaction without rolling back the parent. Unlike the `wait()` method or explicit condition variables, `retry` does not easily lend itself to lost wake-up bugs.

Recall that waiting for one of several conditions to become *true* was impossible using objects with internal monitor condition variables. A novel aspect of `retry` is that such composition becomes easy. Fig. 18.8 shows a code snippet illustrating the `orElse` statement, which joins two or more code blocks. Here, the thread executes the first block (Line 2). If that block calls `retry`, then that subtransaction is rolled back, and the thread executes the second block (Line 4). If that block also calls `retry`, then the `orElse` as a whole pauses, and later reruns each of the blocks (when something changes) until one completes.

In the rest of this chapter, we examine techniques for implementing transactional memory. Transactional synchronization can be implemented in hardware (*HTM*), in software (*STM*), or both. In the following sections, we examine STM implementations.