8 Monitors and Blocking Synchronization

8.1 Introduction

Monitors are a structured way of combining synchronization and data. A class encapsulates both data and methods in the same way that a monitor combines data, methods, and synchronization in a single modular package.

Here is why modular synchronization is important. Let us imagine our application has two threads, a producer and a consumer, that communicate through a shared FIFO queue. We could have the threads share two objects: an unsynchronized queue, and a lock to protect the queue. The producer looks something like this:

```
mutex.lock();
try {
  queue.enq(x)
} finally {
  mutex.unlock();
}
```

This is no way to run a railroad. Suppose the queue is bounded, meaning that an attempt to add an item to a full queue cannot proceed until the queue has room. Here, the decision whether to block the call or to let it proceed depends on the queue's internal state, which is (and should be) inaccessible to the caller. Even worse, suppose the application grows to have multiple producers, consumers, or both. Each such thread must keep track of both the lock and the queue objects, and the application will be correct only if each thread follows the same locking conventions.

A more sensible approach is to allow each queue to manage its own synchronization. The queue itself has its own internal lock, acquired by each method when it is called and released when it returns. There is no need to ensure that every thread that uses the queue follows a cumbersome synchronization protocol. If a thread tries to enqueue an item to a queue that is already full, then the `enq()` method itself can detect the problem, suspend the caller, and resume the caller when the queue has room.

The Art of Multiprocessor Programming. DOI: 10.1016/B978-0-12-397337-5.00008-3

8.2 Monitor Locks and Conditions

Just as in Chapters 2 and 7, a `Lock` is the basic mechanism for ensuring mutual exclusion. Only one thread at a time can *hold* a lock. A thread *acquires* a lock when it first starts to hold the lock. A thread *releases* a lock when it stops holding the lock. A monitor exports a collection of methods, each of which acquires the lock when it is called, and releases it when it returns.

If a thread cannot immediately acquire a lock, it can either *spin*, repeatedly testing whether the desired event has happened, or it can *block*, giving up the processor for a while to allow another thread to run.[1] Spinning makes sense on a multiprocessor if we expect to wait for a short time, because blocking a thread requires an expensive call to the operating system. On the other hand, blocking makes sense if we expect to wait for a long time, because a spinning thread keeps a processor busy without doing any work.

For example, a thread waiting for another thread to release a lock should spin if that particular lock is held briefly, while a consumer thread waiting to dequeue an item from an empty buffer should block, since there is usually no way to predict how long it may have to wait. Often, it makes sense to combine spinning and blocking: a thread waiting to dequeue an item might spin for a brief duration, and then switch to blocking if the delay appears to be long. Blocking works on both multiprocessors and uniprocessors, while spinning works only on multiprocessors.

Pragma 8.2.1. Most of the locks in this book follow the interface shown in Fig. 8.1. Here is an explanation of the `Lock` interface's methods:

- The `lock()` method blocks the caller until it acquires the lock.
- The `lockInterruptibly()` method acts like `lock()`, but throws an exception if the thread is interrupted while it is waiting. (See Pragma 8.2.2.)
- The `unlock()` method releases the lock.
- The `newCondition()` method is a *factory* that creates and returns a `Condition` object associated with the lock (explained below.)
- The `tryLock()` method acquires the lock if it is free, and immediately returns a Boolean indicating whether it acquired the lock. This method can also be called with a timeout.

1 Elsewhere we make a distinction between blocking and nonblocking synchronization algorithms. There, we mean something entirely different: a blocking algorithm is one where a delay by one thread can cause a delay in another.

```
1   public interface Lock {
2     void lock();
3     void lockInterruptibly() throws InterruptedException;
4     boolean tryLock();
5     boolean tryLock(long time, TimeUnit unit);
6     Condition newCondition();
7     void unlock();
8   }
```

Figure 8.1 The Lock Interface.

8.2.1 Conditions

While a thread is waiting for something to happen, say, for another thread to place an item in a queue, it is a very good idea to release the lock on the queue, because otherwise the other thread will never be able to enqueue the anticipated item. After the waiting thread has released the lock, it needs a way to be notified when to reacquire the lock and try again.

In the Java concurrency package (and in related packages such as Pthreads), the ability to release a lock temporarily is provided by a `Condition` object associated with a lock. Fig. 8.2 shows the use of the `Condition` interface provided in the java.util.concurrent.locks library. A condition is associated with a lock, and is created by calling that lock's `newCondition()` method. If the thread holding that lock calls the associated condition's `await()` method, it releases that lock and suspends itself, giving another thread the opportunity to acquire the lock. When the calling thread awakens, it reacquires the lock, perhaps competing with other threads.

Pragma 8.2.2. Threads in Java can be *interrupted* by other threads. If a thread is interrupted during a call to a `Condition`'s `await()` method, then the call throws `InterruptedException`. The proper response to an interrupt is application-dependent. (It is not good programming practice simply to ignore interrupts).
Fig. 8.2 shows a schematic example

```
1   Condition condition = mutex.newCondition();
2   ...
3   mutex.lock()
4   try {
5     while (!property) { // not happy
6       condition.await(); // wait for property
7     }   catch (InterruptedException e) {
8           ... // application-dependent response
9           }
10    ... // happy: property must hold
11  }
```

Figure 8.2 How to use `Condition` objects.

To avoid clutter, we usually omit `InterruptedException` handlers from example code, even though they would be required in actual code.

Like locks, `Condition` objects must be used in a stylized way. Suppose a thread wants to wait until a certain property holds. The thread tests the property while holding the lock. If the property does not hold, then the thread calls `await()` to release the lock and sleep until it is awakened by another thread. Here is the key point: there is no guarantee that the property will hold at the time the thread awakens. The `await()` method can return spuriously (i.e., for no reason), or the thread that signaled the condition may have awakened too many sleeping threads. Whatever the reason, the thread must retest the property, and if it finds the property still does not hold, it must call `await()` again.

The `Condition` interface in Fig. 8.3 provides several variations of this call, some of which provide the ability to specify a maximum time the caller can be suspended, or whether the thread can be interrupted while it is waiting. When the queue changes, the thread that made the change can notify other threads waiting on a condition. Calling `signal()` wakes up one thread waiting on a condition, while calling `signalAll()` wakes up all waiting threads. Fig. 8.4 describes a schematic execution of a monitor lock.

Fig. 8.5 shows how to implement a bounded FIFO queue using explicit locks and conditions. The `lock` field is a lock that must be acquired by all methods. We must initialize it to hold an instance of a class that implements the `Lock` interface. Here, we choose `ReentrantLock`, a useful lock type provided by the java.util.concurrent.locks package. As discussed in Section 8.4, this lock is *reentrant*: a thread that is holding the lock can acquire it again without blocking.

There are two condition objects: `notEmpty` notifies waiting dequeuers when the queue goes from being empty to nonempty, and `notFull` for the opposite direction. Using two conditions instead of one is more efficient, since fewer threads are woken up unnecessarily, but it is more complex.

```
1  public interface Condition {
2    void await() throws InterruptedException;
3    boolean await(long time, TimeUnit unit)
4      throws InterruptedException;
5    boolean awaitUntil(Date deadline)
6      throws InterruptedException;
7    long awaitNanos(long nanosTimeout)
8      throws InterruptedException;
9    void awaitUninterruptibly();
10   void signal();      // wake up one waiting thread
11   void signalAll();   // wake up all waiting threads
12 }
```

Figure 8.3 The `Condition` interface: `await()` and its variants release the lock, give up the processor, and later awaken and reacquire the lock. The `signal()` and `signalAll()` methods awaken one or more waiting threads.

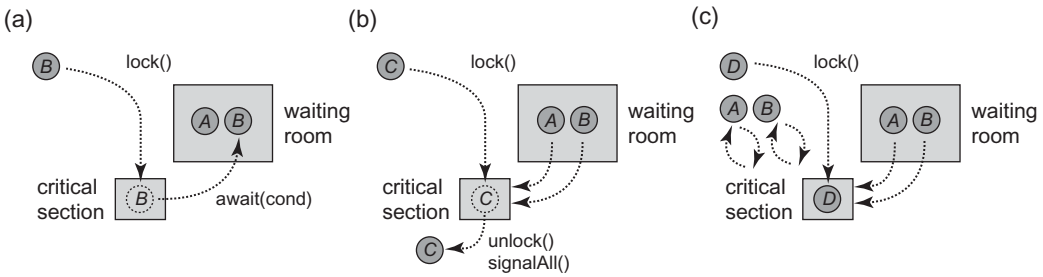

Figure 8.4 A schematic representation of a monitor execution. In Part (a) thread *A* has acquired the monitor lock, called `await()` on a condition, released the lock, and is now in the waiting room. Thread *B* then goes through the same sequence of steps, entering the critical section, calling `await()` on the condition, relinquishing the lock and entering the waiting room. In Part (b) both *A* and *B* leave the waiting room after thread *C* exits the critical section and calls `signalAll()`. *A* and *B* then attempt to reacquire the monitor lock. However, thread *D* manages to acquire the critical section lock first, and so both *A* and *B* spin until *D* leaves the critical section. Notice that if *C* would have issued a `signal()` instead of a `signalAll()`, only one of *A* or *B* would have left the waiting room, and the other would have continued to wait.

This combination of methods, mutual exclusion locks, and condition objects is called a *monitor*.

8.2.2 The Lost-Wakeup Problem

Just as locks are inherently vulnerable to deadlock, `Condition` objects are inherently vulnerable to *lost wakeups*, in which one or more threads wait forever without realizing that the condition for which they are waiting has become true.

Lost wakeups can occur in subtle ways. Fig. 8.6 shows an ill-considered optimization of the `Queue<T>` class. Instead of signaling the `notEmpty` condition each time `enq()` enqueues an item, would it not be more efficient to signal the condition only when the queue actually transitions from empty to non-empty? This optimization works as intended if there is only one producer and one consumer, but it is incorrect if there are multiple producers or consumers. Consider the following scenario: consumers *A* and *B* both try to dequeue an item from an empty queue, both detect the queue is empty, and both block on the `notEmpty` condition. Producer *C* enqueues an item in the buffer, and signals `notEmpty`, waking *A*. Before *A* can acquire the lock, however, another producer *D* puts a second item in the queue, and because the queue is not empty, it does not signal `notEmpty`. Then *A* acquires the lock, removes the first item, but *B*, victim of a lost wakeup, waits forever even though there is an item in the buffer to be consumed.

Although there is no substitute for reasoning carefully about our program, there are simple programming practices that will minimize vulnerability to lost wakeups.

- Always signal *all* processes waiting on a condition, not just one.
- Specify a timeout when waiting.

```
1  class LockedQueue<T> {
2    final Lock lock = new ReentrantLock();
3    final Condition notFull = lock.newCondition();
4    final Condition notEmpty = lock.newCondition();
5    final T[] items;
6    int tail, head, count;
7    public LockedQueue(int capacity) {
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) {
11     lock.lock();
12     try {
13       while (count == items.length)
14         notFull.await();
15       items[tail] = x;
16       if (++tail == items.length)
17         tail = 0;
18       ++count;
19       notEmpty.signal();
20     } finally {
21       lock.unlock();
22     }
23   }
24   public T deq() {
25     lock.lock();
26     try {
27       while (count == 0)
28         notEmpty.await();
29       T x = items[head];
30       if (++head == items.length)
31         head = 0;
32       --count;
33       notFull.signal();
34       return x;
35     } finally {
36       lock.unlock();
37     }
38   }
39 }
```

Figure 8.5 The `LockedQueue` class: a FIFO queue using locks and conditions. There are two condition fields, one to detect when the queue becomes nonempty, and one to detect when it becomes nonfull.

Either of these two practices would fix the bounded buffer error we just described. Each has a small performance penalty, but negligible compared to the cost of a lost wakeup.

Java provides built-in support for monitors in the form of **`synchronized`** blocks and methods, as well as built-in `wait()`, `notify()`, and `notifyAll()` methods. (See Appendix A.)

```
1    public void enq(T x) {
2      lock.lock();
3      try {
4        while (count == items.length)
5          notFull.await();
6        items[tail] = x;
7        if (++tail == items.length)
8          tail = 0;
9        ++count;
10       if (count == 1) {  // Wrong!
11         notEmpty.signal();
12       }
13     } finally {
14       lock.unlock();
15     }
16   }
```

Figure 8.6 This example is *incorrect*. It suffers from lost wakeups. The `enq()` method signals `notEmpty` only if it is the first to place an item in an empty buffer. A lost wakeup occurs if multiple consumers are waiting, but only the first is awakened to consume an item.

8.3 Readers–Writers Locks

Many shared objects have the property that most method calls, called *readers*, return information about the object's state without modifying the object, while only a small number of calls, called *writers*, actually modify the object.

There is no need for readers to synchronize with one another; it is perfectly safe for them to access the object concurrently. Writers, on the other hand, must lock out readers as well as other writers. A *readers–writers* lock allows multiple readers or a single writer to enter the critical section concurrently. We use the following interface:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

This interface exports two lock objects: the *read lock* and the *write lock*. They satisfy the following safety properties:

- No thread can acquire the write lock while any thread holds either the write lock or the read lock.
- No thread can acquire the read lock while any thread holds the write lock.

Naturally, multiple threads may hold the read lock at the same time.

8.3.1 Simple Readers–Writers Lock

We consider a sequence of increasingly sophisticated reader–writer lock implementations. The `SimpleReadWriteLock` class appears in Figs. 8.7– 8.9. This class uses a counter to keep track of the number of readers that have acquired the lock,

```
1  public class SimpleReadWriteLock implements ReadWriteLock {
2    int readers;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public SimpleReadWriteLock() {
8      writer = false;
9      readers = 0;
10     lock = new ReentrantLock();
11     readLock = new ReadLock();
12     writeLock = new WriteLock();
13     condition = lock.newCondition();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
```

Figure 8.7 The `SimpleReadWriteLock` class: fields and public methods.

```
21   class ReadLock implements Lock {
22     public void lock() {
23       lock.lock();
24       try {
25         while (writer) {
26           condition.await();
27         }
28         readers++;
29       } finally {
30         lock.unlock();
31       }
32     }
33     public void unlock() {
34       lock.lock();
35       try {
36         readers--;
37         if (readers == 0)
38           condition.signalAll();
39       } finally {
40         lock.unlock();
41       }
42     }
43   }
```

Figure 8.8 The `SimpleReadWriteLock` class: the inner read lock.

```
44    protected class WriteLock implements Lock {
45      public void lock() {
46        lock.lock();
47        try {
48          while (readers > 0 || writer) {
49            condition.await();
50          }
51          writer = true;
52        } finally {
53          lock.unlock();
54        }
55      }
56      public void unlock() {
57        lock.lock();
58        try {
59         writer = false;
60         condition.signalAll();
61        } finally {
62          lock.unlock();
63        }
64      }
65    }
66  }
```

Figure 8.9 The `SimpleReadWriteLock` class: inner write lock.

and a Boolean field indicating whether a writer has acquired the lock. To define the associated read–write locks, this code uses *inner classes*, a Java feature that allows one object (the `SimpleReadWriteLock` lock) to create other objects (the read–write locks) that share the first object's private fields. Both the `readLock()` and the `writeLock()` methods return objects that implement the `Lock` interface. These objects communicate via the `writeLock()` object's fields. Because the read–write lock methods must synchronize with one another, they both synchronize on the `Lock` and `condition` fields of their common `SimpleReadWriteLock` object.

8.3.2 Fair Readers–Writers Lock

Even though the `SimpleReadWriteLock` algorithm is correct, it is still not quite satisfactory. If readers are much more frequent than writers, as is usually the case, then writers could be locked out for a long time by a continual stream of readers. The `FifoReadWriteLock` class, shown in Figs. 8.10–8.12, shows one way to give writers priority. This class ensures that once a writer calls the write lock's `lock()` method, then no more readers will be able to acquire the read lock until the writer has acquired and released the write lock. Eventually, the readers holding the read lock will drain out without letting any more readers in, and the writer will acquire the write lock.

The `readAcquires` field counts the total number of read lock acquisitions, and the `readReleases` field counts the total number of read lock releases. When these quantities match, no thread is holding the read lock. (We are, of course, ignoring potential integer overflow and wraparound problems.) The class has

```
1  public class FifoReadWriteLock implements ReadWriteLock {
2    int readAcquires, readReleases;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public FifoReadWriteLock() {
8      readAcquires = readReleases = 0;
9      writer   = false;
10     lock = new ReentrantLock(true);
11     condition = lock.newCondition();
12     readLock = new ReadLock();
13     writeLock = new WriteLock();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
21   ...
22 }
```

Figure 8.10 The FifoReadWriteLock class: fields and public methods.

```
23   private class ReadLock implements Lock {
24     public void lock() {
25       lock.lock();
26       try {
27         while (writer) {
28           condition.await();
29         }
30         readAcquires++;
31       } finally {
32         lock.unlock();
33       }
34     }
35     public void unlock() {
36       lock.lock();
37       try {
38         readReleases++;
39         if (readAcquires == readReleases)
40           condition.signalAll();
41       } finally {
42         lock.unlock();
43       }
44     }
45   }
```

Figure 8.11 The FifoReadWriteLock class: inner read lock class.

```
46    private class WriteLock implements Lock {
47      public void lock() {
48        lock.lock();
49        try {
50          while (writer) {
51            condition.await();
52          }
53          writer = true;
54          while (readAcquires != readReleases) {
55            condition.await();
56          }
57        } finally {
58          lock.unlock();
59        }
60      }
61      public void unlock() {
62        writer = false;
63        condition.signalAll();
64      }
65    }
```

Figure 8.12 The FifoReadWriteLock class: inner write lock class.

a private lock field, held by readers for short durations: they acquire the lock, increment the readAcquires field, and release the lock. Writers hold this lock from the time they try to acquire the write lock to the time they release it. This locking protocol ensures that once a writer has acquired the lock, no additional reader can increment readAcquires, so no additional reader can acquire the read lock, and eventually all readers currently holding the read lock will release it, allowing the writer to proceed.

How are waiting writers notified when the last reader releases its lock? When a writer tries to acquire the write lock, it acquires the FifoReadWriteLock object's lock. A reader releasing the read lock also acquires that lock, and calls the associated condition's signal() method if all readers have released their locks.

8.4 Our Own Reentrant Lock

Using the locks described in Chapters 2 and 7, a thread that attempts to reacquire a lock it already holds will deadlock with itself. This situation can arise if a method that acquires a lock makes a nested call to another method that acquires the same lock.

A lock is *reentrant* if it can be acquired multiple times by the same thread. We now examine how to create a reentrant lock from a non-reentrant lock. This exercise is intended to illustrate how to use locks and conditions. In practice, the java.util.concurrent.locks package provides reentrant lock classes, so there is no need to write our own.

Fig. 8.13 shows the SimpleReentrantLock class. The owner field holds the ID of the last thread to acquire the lock, and the holdCount field is incremented each

```
1  public class SimpleReentrantLock implements Lock{
2    Lock lock;
3    Condition condition;
4    int owner, holdCount;
5    public SimpleReentrantLock() {
6      lock = new SimpleLock();
7      condition = lock.newCondition();
8      owner = 0;
9      holdCount = 0;
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     lock.lock();
14     try {
15       if (owner == me) {
16         holdCount++;
17         return;
18       }
19       while (holdCount != 0) {
20        condition.await();
21       }
22       owner = me;
23       holdCount = 1;
24    } finally {
25      lock.unlock();
26    }
27   }
28   public void unlock() {
29     lock.lock();
30     try {
31       if (holdCount == 0 || owner != ThreadID.get())
32         throw new IllegalMonitorStateException();
33       holdCount--;
34       if (holdCount == 0) {
35         condition.signal();
36       }
37     } finally {
38       lock.unlock();
39     }
40   }
41
42   public Condition newCondition() {
43     throw new UnsupportedOperationException("Not supported yet.");
44   }
45  ...
46 }
```

Figure 8.13 The `SimpleReentrantLock` class: `lock()` and `unlock()` methods.

time the lock is acquired, and decremented each time it is released. The lock is free when the `holdCount` value is zero. Because these two fields are manipulated atomically, we need an internal, short-term lock. The `lock` field is a lock used by `lock()` and `unlock()` to manipulate the fields, and the `condition` field is used by

threads waiting for the lock to become free. In Fig. 8.13, we initialize the internal `lock` field to an object of a (fictitious) `SimpleLock` class which is presumably not reentrant (Line 6).

The `lock`() method acquires the internal lock (Line 13). If the current thread is already the owner, it increments the hold count and returns (Line 15). Otherwise, if the hold count is not zero, the lock is held by another thread, and the caller releases the lock and waits until the condition is signaled (Line 20). When the caller awakens, it must still check that the hold count is zero. When the hold count is established to be zero, the calling thread makes itself the owner and sets the hold count to 1.

The `unlock`() method acquires the internal lock (Line 29). It throws an exception if either the lock is free, or the caller is not the owner (Line 31). Otherwise, it decrements the hold count. If the hold count is zero, then the lock is free, so the caller signals the condition to wake up a waiting thread (Line 35).

8.5 Semaphores

As we have seen, a mutual exclusion lock guarantees that only one thread at a time can enter a critical section. If another thread wants to enter the critical section while it is occupied, then it blocks, suspending itself until another thread notifies it to try again. A `Semaphore` is a generalization of mutual exclusion locks. Each `Semaphore` has a *capacity*, denoted by c for brevity. Instead of allowing only one thread at a time into the critical section, a `Semaphore` allows at most c threads, where the capacity c is determined when the `Semaphore` is initialized. As discussed in the chapter notes, semaphores were one of the earliest forms of synchronization.

The `Semaphore` class of Fig. 8.14 provides two methods: a thread calls `acquire`() to request permission to enter the critical section, and `release`() to announce that it is leaving the critical section. The `Semaphore` itself is just a counter: it keeps track of the number of threads that have been granted permission to enter. If a new `acquire`() call is about to exceed the capacity c, the calling thread is suspended until there is room. When a thread leaves the critical section, it calls `release`() to notify a waiting thread that there is now room.

8.6 Chapter Notes

Monitors were invented by Per Brinch-Hansen [52] and Tony Hoare [71]. Semaphores were invented by Edsger Dijkstra [33]. McKenney [112] surveys different kinds of locking protocols.

```
1  public class Semaphore {
2    final int capacity;
3    int state;
4    Lock lock;
5    Condition condition;
6    public Semaphore(int c) {
7      capacity = c;
8      state = 0;
9      lock = new ReentrantLock();
10     condition = lock.newCondition();
11   }
12   public void acquire() {
13     lock.lock();
14     try {
15       while (state == capacity) {
16         condition.await();
17       }
18       state++;
19     } finally {
20       lock.unlock();
21     }
22   }
23   public void release() {
24     lock.lock();
25     try {
26       state--;
27       condition.signalAll();
28     } finally {
29       lock.unlock();
30     }
31   }
32 }
```

Figure 8.14 Semaphore implementation.

8.7 Exercises

Exercise 93. Reimplement the `SimpleReadWriteLock` class using Java **`synchronized`**, `wait()`, `notify()`, and `notifyAll()` constructs in place of explict locks and conditions.

Hint: you must figure out how methods of the inner read–write lock classes can lock the outer `SimpleReadWriteLock` object.

Exercise 94. The `ReentrantReadWriteLock` class provided by the java.util.concurrent.locks package does not allow a thread holding the lock in read mode to then access that lock in write mode (the thread will block). Justify this design decision by sketching what it would take to permit such lock upgrades.

Exercise 95. A *savings account* object holds a nonnegative balance, and provides `deposit(`k`)` and `withdraw(`k`)` methods, where `deposit(`k`)` adds k to the balance, and `withdraw(`k`)` subtracts k, if the balance is at least k, and otherwise blocks until the balance becomes k or greater.

1. Implement this savings account using locks and conditions.
2. Now suppose there are two kinds of withdrawals: *ordinary* and *preferred*. Devise an implementation that ensures that no ordinary withdrawal occurs if there is a preferred withdrawal waiting to occur.
3. Now let us add a `transfer`() method that transfers a sum from one account to another:

```
void transfer(int k, Account reserve) {
  lock.lock();
  try {
    reserve.withdraw(k);
    deposit(k);
  } finally {lock.unlock();}
}
```

   We are given a set of 10 accounts, whose balances are unknown. At 1:00, each of n threads tries to transfer $100 from another account into its own account. At 2:00, a Boss thread deposits $1000 to each account. Is every transfer method called at 1:00 certain to return?

Exercise 96. In the *shared bathroom problem*, there are two classes of threads, called *male* and *female*. There is a single *bathroom* resource that must be used in the following way:

1. Mutual exclusion: persons of opposite sex may not occupy the bathroom simultaneously,
2. Starvation-freedom: everyone who needs to use the bathroom eventually enters.

The protocol is implemented via the following four procedures: `enterMale`() delays the caller until it is ok for a male to enter the bathroom, `leaveMale`() is called when a male leaves the bathroom, while `enterFemale`() and `leaveFemale`() do the same for females. For example,

```
enterMale();
teeth.brush(toothpaste);
leaveMale();
```

1. Implement this class using locks and condition variables.
2. Implement this class using **`synchronized`**, `wait`(), `notify`(), and `notifyAll`().

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

Exercise 97. The `Rooms` class manages a collection of *rooms*, indexed from 0 to *m* (where *m* is an argument to the constructor). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. For example, if there are two rooms, indexed 0 and 1, then any number of threads might enter the room 0, but no thread can enter the room 1 while room 0 is occupied. Fig. 8.15 shows an outline of the `Rooms` class.

Each room can be assigned an *exit handler*: calling `setHandler`(i, h) sets the exit handler for room i to handler h. The exit handler is called by the last thread to

```
1  public class Rooms {
2    public interface Handler {
3      void onEmpty();
4    }
5    public Rooms(int m) { ... };
6    void enter(int i) { ... };
7    boolean exit() { ... };
8    public void setExitHandler(int i, Rooms.Handler h) { ... };
9  }
```

Figure 8.15 The `Rooms` class.

```
1  class Driver {
2    void main() {
3      CountDownLatch startSignal = new CountDownLatch(1);
4      CountDownLatch doneSignal = new CountDownLatch(n);
5      for (int i = 0; i < n; ++i) // start threads
6        new Thread(new Worker(startSignal, doneSignal)).start();
7      doSomethingElse();          // get ready for threads
8      startSignal.countDown();    // unleash threads
9      doSomethingElse();          // biding my time ...
10     doneSignal.await();         // wait for threads to finish
11   }
12   class Worker implements Runnable {
13     private final CountDownLatch startSignal, doneSignal;
14     Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
15       startSignal = myStartSignal;
16       doneSignal = myDoneSignal;
17     }
18     public void run() {
19       startSignal.await();    // wait for driver's OK to start
20       doWork();
21       doneSignal.countDown(); // notify driver we're done
22     }
23     ...
24   }
25 }
```

Figure 8.16 The `CountDownLatch` class: an example usage.

leave a room, but before any threads subsequently enter any room. This method is called once and while it is running, no threads are in any rooms.

Implement the `Rooms` class. Make sure that:

- If some thread is in room i, then no thread is in room $j \neq i$.
- The last thread to leave a room calls the room's exit handler, and no threads are in any room while that handler is running.
- Your implementation must be *fair*: any thread that tries to enter a room eventually succeeds. Naturally, you may assume that every thread that enters a room eventually leaves.

Exercise 98. Consider an application with distinct sets of *active* and *passive* threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A `CountDownLatch` encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls `countDown()`, which decrements the counter. Each passive thread calls `await()`, which blocks the thread until the counter reaches zero. (See Fig. 8.16.)

Provide a `CountDownLatch` implementation. Do not worry about reusing the `CountDownLatch` object.

Exercise 99. This exercise is a follow-up to Exercise 98. Provide a `CountDownLatch` implementation where the `CountDownLatch` object can be reused.

This page intentionally left blank

9 Linked Lists: The Role of Locking

9.1 Introduction

In Chapter 7 we saw how to build scalable spin locks that provide mutual exclusion efficiently, even when they are heavily used. We might think that it is now a simple matter to construct scalable concurrent data structures: take a sequential implementation of the class, add a scalable lock field, and ensure that each method call acquires and releases that lock. We call this approach *coarse-grained synchronization*.

Often, coarse-grained synchronization works well, but there are important cases where it does not. The problem is that a class that uses a single lock to mediate all its method calls is not always scalable, even if the lock itself is scalable. Coarse-grained synchronization works well when levels of concurrency are low, but if too many threads try to access the object at the same time, then the object becomes a sequential bottleneck, forcing threads to wait in line for access.

This chapter introduces several useful techniques that go beyond coarse-grained locking to allow multiple threads to access a single object at the same time.

- *Fine-grained synchronization:* Instead of using a single lock to synchronize every access to an object, we split the object into independently synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time.
- *Optimistic synchronization:* Many objects, such as trees or lists, consist of multiple components linked together by references. Some methods search for a particular component (e.g., a list or tree node containing a particular key). One way to reduce the cost of fine-grained locking is to search without acquiring any locks at all. If the method finds the sought-after component, it locks that component, and then checks that the component has not changed in the interval between when it was inspected and when it was locked. This technique is worthwhile only if it succeeds more often than not, which is why we call it optimistic.

The Art of Multiprocessor Programming. DOI: 10.1016/B978-0-12-397337-5.00009-5

```
1  public interface Set<T> {
2    boolean add(T x);
3    boolean remove(T x);
4    boolean contains(T x);
5  }
```

Figure 9.1 The Set interface: add() adds an item to the set (no effect if that item is already present), remove() removes it (if present), and contains() returns a Boolean indicating whether the item is present.

- *Lazy synchronization:* Sometimes it makes sense to postpone hard work. For example, the task of removing a component from a data structure can be split into two phases: the component is *logically removed* simply by setting a tag bit, and later, the component can be *physically removed* by unlinking it from the rest of the data structure.
- *Nonblocking synchronization:* Sometimes we can eliminate locks entirely, relying on built-in atomic operations such as `compareAndSet()` for synchronization.

Each of these techniques can be applied (with appropriate customization) to a variety of common data structures. In this chapter we consider how to use linked lists to implement a *set*, a collection of *items* that contains no duplicate elements.

For our purposes, as illustrated in Fig. 9.1, a *set* provides the following three methods:

- The `add`(x) method adds x to the set, returning *true* if, and only if x was not already there.
- The `remove`(x) method removes x from the set, returning *true* if, and only if x was there.
- The `contains`(x) returns *true* if, and only if the set contains x.

For each method, we say that a call is *successful* if it returns *true*, and *unsuccessful* otherwise. It is typical that in applications using sets, there are significantly more `contains`() calls than `add`() or `remove`() calls.

9.2 List-Based Sets

This chapter presents a range of concurrent set algorithms, all based on the same basic idea. A set is implemented as a linked list of nodes. As shown in Fig. 9.2, the `Node<T>` class has three fields.[1] The `item` field is the actual item of interest. The `key` field is the item's hash code. Nodes are sorted in `key` order, providing an efficient way to detect when an item is absent. The `next` field is a reference to

1 To be consistent with the Java memory model these fields and their modifications later in the text will need to be volatile, though we ignore this issue here for the sake of brevity.

```
1  private class Node {
2    T item;
3    int key;
4    Node next;
5  }
```

Figure 9.2 The `Node<T>` class: this internal class keeps track of the item, the item's key, and the next node in the list. Some algorithms require technical changes to this class.

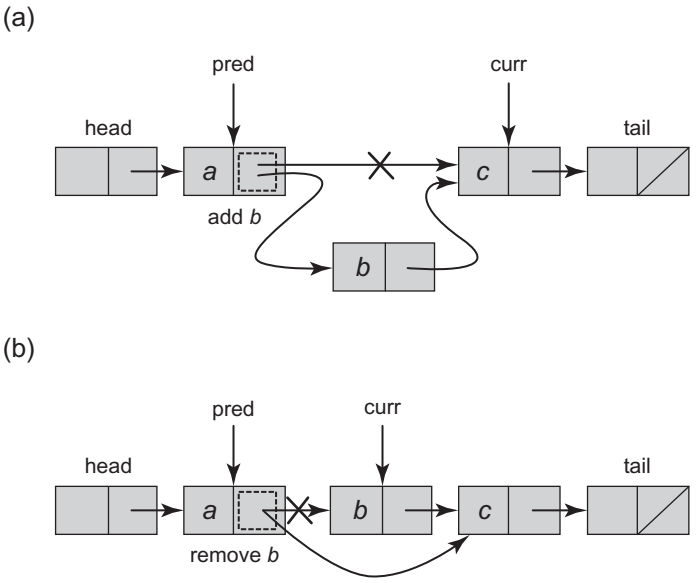

Figure 9.3 A seqential `Set` implementation: adding and removing nodes. In Part (a), a thread adding a node *b* uses two variables: `curr` is the current node, and `pred` is its predecessor. The thread moves down the list comparing the keys for `curr` and *b*. If a match is found, the item is already present, so it returns false. If `curr` reaches a node with a higher key, the item is not in the set so Set *b*'s `next` field to `curr`, and `pred`'s `next` field to *b*. In Part (b), to delete `curr`, the thread sets `pred`'s `next` field to `curr`'s `next` field.

the next node in the list. (Some of the algorithms we consider require technical changes to this class, such as adding new fields, or changing the types of existing fields.) For simplicity, we assume that each item's hash code is unique (relaxing this assumption is left as an exercise). We associate an item with the same node and key throughout any given example, which allows us to abuse notation and use the same symbol to refer to a node, its key, and its item. That is, node *a* may have key *a* and item *a*, and so on.

The list has two kinds of nodes. In addition to *regular* nodes that hold items in the set, we use two *sentinel* nodes, called `head` and `tail`, as the first and last list elements. Sentinel nodes are never added, removed, or searched for, and their

keys are the minimum and maximum integer values.[2] Ignoring synchronization for the moment, the top part of Fig. 9.3 schematically describes how an item is added to the set. Each thread A has two local variables used to traverse the list: $\texttt{curr}_A$ is the current node and $\texttt{pred}_A$ is its predecessor. To add an item to the set, thread A sets local variables $\texttt{pred}_A$ and $\texttt{curr}_A$ to `head`, and moves down the list, comparing $\texttt{curr}_A$'s key to the key of the item being added. If they match, the item is already present in the set, so the call returns *false*. If $\texttt{pred}_A$ precedes $\texttt{curr}_A$ in the list, then $\texttt{pred}_A$'s key is lower than that of the inserted item, and $\texttt{curr}_A$'s key is higher, so the item is not present in the list. The method creates a new node b to hold the item, sets b's $\texttt{next}_A$ field to $\texttt{curr}_A$, then sets $\texttt{pred}_A$ to b. Removing an item from the set works in a similar way.

9.3 Concurrent Reasoning

Reasoning about concurrent data structures may seem impossibly difficult, but it is a skill that can be learned. Often, the key to understanding a concurrent data structure is to understand its *invariants*: properties that always hold. We can show that a property is invariant by showing that:

1. The property holds when the object is created, and
2. Once the property holds, then no thread can take a step that makes the property *false*.

Most interesting invariants hold trivially when the list is created, so it makes sense to focus on how invariants, once established, are preserved.

Specifically, we can check that each invariant is preserved by each invocation of `insert()`, `remove()`, and `contains()` methods. This approach works only if we can assume that these methods are the *only* ones that modify nodes, a property sometimes called *freedom from interference*. In the list algorithms considered here, nodes are internal to the list implementation, so freedom from interference is guaranteed because users of the list have no opportunity to modify its internal nodes.

We require freedom from interference even for nodes that have been removed from the list, since some of our algorithms permit a thread to unlink a node while it is being traversed by others. Fortunately, we do not attempt to reuse list nodes that have been removed from the list, relying instead on a garbage collector to recycle that memory. The algorithms described here work in languages without garbage collection, but sometimes require nontrivial modifications that are beyond the scope of this chapter.

---

2 All algorithms presented here work for any ordered set of keys that have maximum and minimum values and that are well-founded, that is, there are only finitely many keys smaller than any given key. For simplicity, we assume here that keys are integers.

When reasoning about concurrent object implementations, it is important to understand the distinction between an object's *abstract value* (here, a set of items), and its *concrete representation* (here, a list of nodes).

Not every list of nodes is a meaningful representation for a set. An algorithm's *representation invariant* characterizes which representations make sense as abstract values. If *a* and *b* are nodes, we say that *a points to b* if *a*'s `next` field is a reference to *b*. We say that *b* is *reachable* if there is a sequence of nodes, starting at `head`, and ending at *b*, where each node in the sequence points to its successor.

The set algorithms in this chapter require the following invariants (some require more, as explained later). First, sentinels are neither added nor removed. Second, nodes are sorted by key, and keys are unique.

Let us think of the representation invariant as a contract among the object's methods. Each method call preserves the invariant, and also relies on the other methods to preserve the invariant. In this way, we can reason about each method in isolation, without having to consider all the possible ways they might interact.

Given a list satisfying the representation invariant, which set does it represent? The meaning of such a list is given by an *abstraction map* carrying lists that satisfy the representation invariant to sets. Here, the abstraction map is simple: an item is in the set if and only if it is reachable from `head`.

What safety and liveness properties do we need? Our safety property is *linearizability*. As we saw in Chapter 3, to show that a concurrent data structure is a linearizable implementation of a sequentially specified object, it is enough to identify a *linearization point*, a single atomic step where the method call "takes effect." This step can be a read, a write, or a more complex atomic operation. Looking at any execution history of a list-based set, it must be the case that if the abstraction map is applied to the representation at the linearization points, the resulting sequence of states and method calls defines a valid sequential set execution. Here, `add`(*a*) adds *a* to the abstract set, `remove`(*a*) removes *a* from the abstract set, and `contains`(*a*) returns *true* or *false*, depending on whether *a* was already in the set.

Different list algorithms make different progress guarantees. Some use locks, and care is required to ensure they are deadlock- and starvation-free. Some *nonblocking* list algorithms do not use locks at all, while others restrict locking to certain methods. Here is a brief summary, from Chapter 3, of the nonblocking properties we use[3]:

- A method is *wait-free* if it guarantees that every call finishes in a finite number of steps.
- A method is *lock-free* if it guarantees that *some* call always finishes in a finite number of steps.

We are now ready to consider a range of list-based set algorithms. We start with algorithms that use coarse-grained synchronization, and successively refine them

3 Chapter 3 introduces an even weaker nonblocking property called *obstruction-freedom.*

to reduce granularity of locking. Formal proofs of correctness lie beyond the scope of this book. Instead, we focus on informal reasoning useful in everyday problem-solving.

As mentioned, in each of these algorithms, methods scan through the list using two local variables: `curr` is the current node and `pred` is its predecessor. These variables are thread-local,[4] so we use $\texttt{pred}_A$ and $\texttt{curr}_A$ to denote the instances used by thread A.

9.4 Coarse-Grained Synchronization

We start with a simple algorithm using coarse-grained synchronization. Figs. 9.4 and 9.5 show the `add()` and `remove()` methods for this coarse-grained algorithm. (The `contains()` method works in much the same way, and is left as an exercise.) The list itself has a single lock which every method call must acquire. The principal

```
1  public class CoarseList<T> {
2    private Node head;
3    private Lock lock = new ReentrantLock();
4    public CoarseList() {
5      head  = new Node(Integer.MIN_VALUE);
6      head.next = new Node(Integer.MAX_VALUE);
7    }
8    public boolean add(T item) {
9      Node pred, curr;
10     int key = item.hashCode();
11     lock.lock();
12     try {
13       pred = head;
14       curr = pred.next;
15       while (curr.key < key) {
16         pred = curr;
17         curr = curr.next;
18       }
19       if (key == curr.key) {
20         return false;
21       } else {
22         Node node = new Node(item);
23         node.next = curr;
24         pred.next = node;
25         return true;
26       }
27     } finally {
28       lock.unlock();
29     }
30   }
```

Figure 9.4 The `CoarseList` class: the `add()` method.

4 Appendix A describes how thread-local variables work in Java.

```
31    public boolean remove(T item) {
32      Node pred, curr;
33      int key = item.hashCode();
34      lock.lock();
35      try {
36        pred = head;
37        curr = pred.next;
38        while (curr.key < key) {
39          pred = curr;
40          curr = curr.next;
41        }
42        if (key == curr.key) {
43          pred.next = curr.next;
44          return true;
45        } else {
46          return false;
47        }
48      } finally {
49        lock.unlock();
50      }
51    }
```

Figure 9.5 The `CoarseList` class: the `remove()` method. All methods acquire a single lock, which is released on exit by the **`finally`** block.

advantage of this algorithm, which should not be discounted, is that it is obviously correct. All methods act on the list only while holding the lock, so the execution is essentially sequential. To simplify matters, we follow the convention (for now) that the linearization point for any method call that acquires a lock is the instant the lock is acquired.

The `CoarseList` class satisfies the same progress condition as its lock: if the `Lock` is starvation-free, so is our implementation. If contention is very low, this algorithm is an excellent way to implement a list. If, however, there is contention, then even if the lock itself performs well, threads will still be delayed waiting for one another.

9.5 Fine-Grained Synchronization

We can improve concurrency by locking individual nodes, rather than locking the list as a whole. Instead of placing a lock on the entire list, let us add a `Lock` to each node, along with `lock()` and `unlock()` methods. As a thread traverses the list, it locks each node when it first visits, and sometime later releases it. Such *fine-grained* locking permits concurrent threads to traverse the list together in a pipelined fashion.

Let us consider two nodes a and b where a points to b. It is not safe to unlock a before locking b because another thread could remove b from the list in the interval between unlocking a and locking b. Instead, thread A must acquire locks in a kind of "hand-over-hand" order: except for the initial `head` sentinel node,

acquire the lock for a node only while holding the lock for its predecessor. This locking protocol is sometimes called *lock coupling*. (Notice that there is no obvious way to implement lock coupling using Java's **synchronized** methods.)

Fig. 9.6 shows the `FineList` algorithm's `add()` method, and Fig. 9.7 its `remove()` method. Just as in the coarse-grained list, `remove()` makes $curr_A$ unreachable by setting $pred_A$'s `next` field to $curr_A$'s successor. To be safe, `remove()` must lock both $pred_A$ and $curr_A$. To see why, let us consider the following scenario, illustrated in Fig. 9.8. Thread *A* is about to remove node *a*, the first node in the list, while thread *B* is about to remove node *b*, where *a* points to *b*. Suppose *A* locks `head`, and *B* locks *a*. *A* then sets `head.next` to *b*, while *B* sets `a.next` to *c*. The net effect is to remove *a*, but not *b*. The problem is that there is no overlap between the locks held by the two `remove()` calls. Fig. 9.9 illustrates how this "hand-over-hand" locking avoids this problem.

To guarantee progress, it is important that all methods acquire locks in the same order, starting at the `head` and following `next` references toward the `tail`. As Fig. 9.10 shows, a deadlock could occur if different method calls were to acquire locks in different orders. In this example, thread *A*, trying to add *a*, has

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

Figure 9.6 The `FineList` class: the `add()` method uses hand-over-hand locking to traverse the list. The **`finally`** blocks release locks before returning.

```
29  public boolean remove(T item) {
30    Node pred = null, curr = null;
31    int key = item.hashCode();
32    head.lock();
33    try {
34      pred = head;
35      curr = pred.next;
36      curr.lock();
37      try {
38        while (curr.key < key) {
39          pred.unlock();
40          pred = curr;
41          curr = curr.next;
42          curr.lock();
43        }
44        if (curr.key == key) {
45          pred.next = curr.next;
46          return true;
47        }
48        return false;
49      } finally {
50        curr.unlock();
51      }
52    } finally {
53      pred.unlock();
54    }
55  }
```

Figure 9.7 The `FineList` class: the `remove()` method locks both the node to be removed and its predecessor before removing that node.

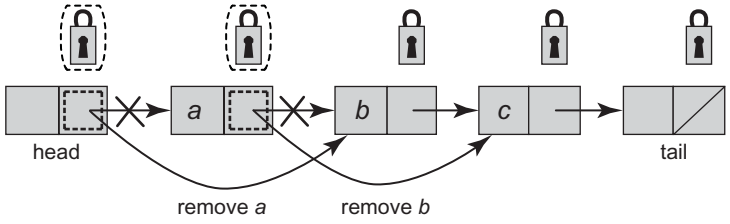

Figure 9.8 The `FineList` class: why `remove()` must acquire two locks. Thread *A* is about to remove *a*, the first node in the list, while thread *B* is about to remove *b*, where *a* points to *b*. Suppose *A* locks `head`, and *B* locks *a*. Thread *A* then sets `head.next` to *b*, while *B* sets *a*'s `next` field to *c*. The net effect is to remove *a*, but not *b*.

locked *b* and is attempting to lock `head`, while *B*, trying to remove *b*, has locked `head` and is trying to lock *b*. Clearly, these method calls will never finish. Avoiding deadlocks is one of the principal challenges of programming with locks.

The `FineList` algorithm maintains the representation invariant: sentinels are never added or removed, and nodes are sorted by key value without duplicates.

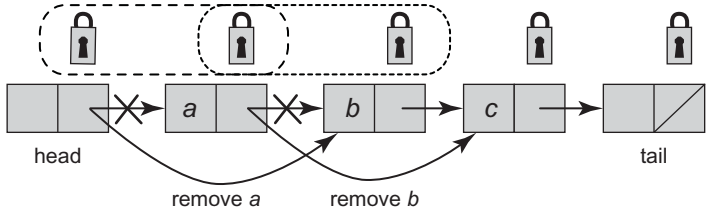

Figure 9.9 The `FineList` class: Hand-over-hand locking ensures that if concurrent `remove()` calls try to remove adjacent nodes, then they acquire conflicting locks. Thread *A* is about to remove node *a*, the first node in the list, while thread *B* is about to remove node *b*, where *a* points to *b*. Because *A* must lock both `head` and *a*, and *B* must lock both *a* and *b*, they are guaranteed to conflict on *a*, forcing one call to wait for the other.

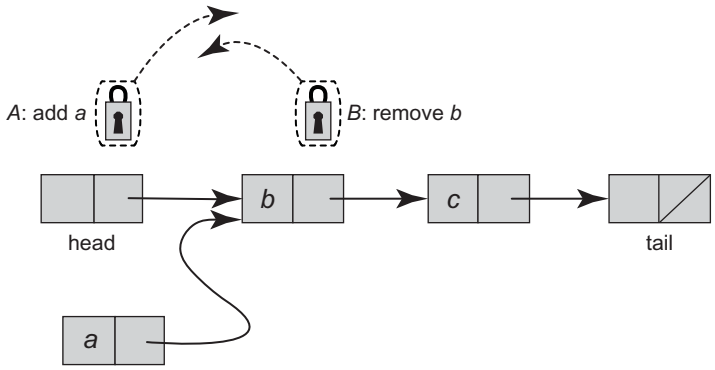

Figure 9.10 The `FineList` class: a deadlock can occur if, for example, `remove()` and `add()` calls acquire locks in opposite order. Thread *A* is about to insert *a* by locking first *b* and then `head`, and thread *B* is about to remove node *b* by locking first `head` and then *b*. Each thread holds the lock the other is waiting to acquire, so neither makes progress.

The abstraction map is the same as for the course-grained list: an item is in the set if, and only if its node is reachable.

The linearization point for an `add`(*a*) call depends on whether the call was successful (i.e., whether *a* was already present). A successful call (*a* absent) is linearized when the node with the next higher key is locked (either Line 7 or 13).

The same distinctions apply to `remove`(*a*) calls. An unsuccessful call (*a* present) is linearized when the predecessor node is locked (Lines 36 or 42). An unsuccessful call (*a* absent) is linearized when the node containing the next higher key is locked (Lines 36 or 42).

Determining linearization points for `contains`() is left as an exercise.

The `FineList` algorithm is starvation-free, but arguing this property is harder than in the coarse-grained case. We assume that all individual locks are starvation-free. Because all methods acquire locks in the same down-the-list

order, deadlock is impossible. If thread A attempts to lock `head`, eventually it succeeds. From that point on, because there are no deadlocks, eventually all locks held by threads ahead of A in the list will be released, and A will succeed in locking $\texttt{pred}_A$ and $\texttt{curr}_A$.

9.6 Optimistic Synchronization

Although fine-grained locking is an improvement over a single, coarse-grained lock, it still imposes a potentially long sequence of lock acquisitions and releases. Moreover, threads accessing disjoint parts of the list may still block one another. For example, a thread removing the second item in the list blocks all concurrent threads searching for later nodes.

One way to reduce synchronization costs is to take a chance: search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct. If a synchronization conflict causes the wrong nodes to be locked, then release the locks and start over. Normally, this kind of conflict is rare, which is why we call this technique *optimistic synchronization*.

In Fig. 9.11, thread A makes an optimistic `add`(a). It traverses the list without acquiring any locks (Lines 6 through 8). In fact, it ignores the locks completely. It stops the traversal when $\texttt{curr}_A$'s key is greater than, or equal to a's. It then locks $\texttt{pred}_A$ and $\texttt{curr}_A$, and calls `validate`() to check that $\texttt{pred}_A$ is reachable and its `next` field still refers to $\texttt{curr}_A$. If validation succeeds, then thread A proceeds as before: if $\texttt{curr}_A$'s key is greater than a, thread A adds a new node with item a between $\texttt{pred}_A$ and $\texttt{curr}_A$, and returns *true*. Otherwise it returns *false*. The `remove`() and `contains`() methods (Figs. 9.12 and 9.13) operate similarly, traversing the list without locking, then locking the target nodes and validating they are still in the list. To be consistent with the Java memory model, the *next* fields in the nodes need to be declared volatile.

The code of `validate`() appears in Fig. 9.14. We are reminded of the following story:

> A tourist takes a taxi in a foreign town. The taxi driver speeds through a red light. The tourist, frightened, asks "What are you are doing?" The driver answers: "Do not worry, I am an expert." He speeds through more red lights, and the tourist, on the verge of hysteria, complains again, more urgently. The driver replies, "Relax, relax, you are in the hands of an expert." Suddenly, the light turns green, the driver slams on the brakes, and the taxi skids to a halt. The tourist picks himself off the floor of the taxi and asks "For crying out loud, why stop now that the light is finally green?" The driver answers "Too dangerous, could be another expert crossing."

Traversing any dynamically changing lock-based data structure while ignoring locks requires careful thought (there are other expert threads out there). We must make sure to use some form of *validation* and guarantee freedom from *interference*.

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4       Node pred = head;
5       Node curr = pred.next;
6       while (curr.key < key) {
7         pred = curr; curr = curr.next;
8       }
9       pred.lock(); curr.lock();
10      try {
11        if (validate(pred, curr)) {
12          if (curr.key == key) {
13            return false;
14          } else {
15            Node node = new Node(item);
16            node.next = curr;
17            pred.next = node;
18            return true;
19          }
20        }
21      } finally {
22        pred.unlock(); curr.unlock();
23      }
24    }
25  }
```

Figure 9.11 The `OptimisticList` class: the `add()` method traverses the list ignoring locks, acquires locks, and validates before adding the new node.

```
26  public boolean remove(T item) {
27    int key = item.hashCode();
28    while (true) {
29      Node pred = head;
30      Node curr = pred.next;
31      while (curr.key < key) {
32        pred = curr; curr = curr.next;
33      }
34      pred.lock(); curr.lock();
35      try {
36        if (validate(pred, curr)) {
37          if (curr.key == key) {
38            pred.next = curr.next;
39            return true;
40          } else {
41            return false;
42          }
43        }
44      } finally {
45        pred.unlock(); curr.unlock();
46      }
47    }
48  }
```

Figure 9.12 The `OptimisticList` class: the `remove()` method traverses ignoring locks, acquires locks, and validates before removing the node.

```
49  public boolean contains(T item) {
50    int key = item.hashCode();
51    while (true) {
52     Node pred = this.head; // sentinel node;
53     Node curr = pred.next;
54     while (curr.key < key) {
55      pred = curr; curr = curr.next;
56      }
57     pred.lock(); curr.lock();
58     try {
59       if (validate(pred, curr)) {
60         return (curr.key == key);
61       }
62     } finally {                  // always unlock
63       pred.unlock(); curr.unlock();
64     }
65    }
66  }
```

Figure 9.13 The `OptimisticList` class: the `contains()` method searches, ignoring locks, then it acquires locks, and validates to determine if the node is in the list.

```
67   private boolean validate(Node pred, Node curr) {
68      Node node = head;
69      while (node.key <= pred.key) {
70        if (node == pred)
71          return pred.next == curr;
72        node = node.next;
73      }
74      return false;
75    }
```

Figure 9.14 The `OptimisticList`: validation checks that $\texttt{pred}_A$ points to $\texttt{curr}_A$ and is reachable from `head`.

As Fig. 9.15 shows, validation is necessary because the trail of references leading to $\texttt{pred}_A$ or the reference from $\texttt{pred}_A$ to $\texttt{curr}_A$ could have changed between when they were last read by A and when A acquired the locks. In particular, a thread could be traversing parts of the list that have already been removed. For example, the node $\texttt{curr}_A$ and all nodes between $\texttt{curr}_A$ and a (including a) may be removed while A is still traversing $\texttt{curr}_A$. Thread A discovers that $\texttt{curr}_A$ points to a, and, without validation, "successfully" removes a, even though a is no longer in the list. A `validate()` call detects that a is no longer in the list, and the caller restarts the method.

Because we are ignoring the locks that protect concurrent modifications, each of the method calls may traverse nodes that have been removed from the list. Nevertheless, absence of interference implies that once a node has been unlinked from the list, the value of its `next` field does not change, so following a sequence of such links eventually leads back to the list. Absence of interference, in turn, relies on garbage collection to ensure that no node is recycled while it is being traversed.

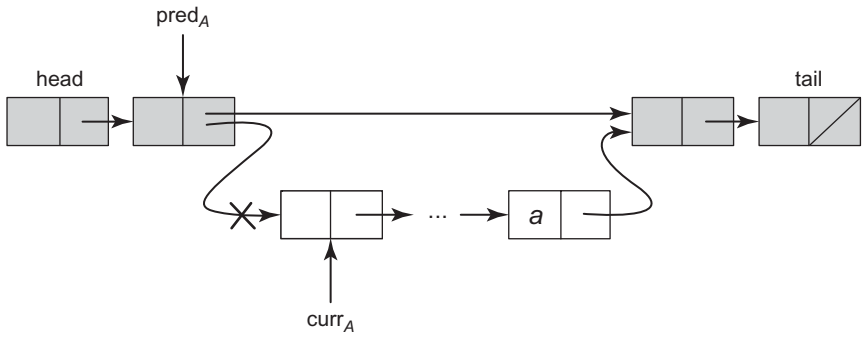

Figure 9.15 The `OptimisticList` class: why validation is needed. Thread *A* is attempting to remove a node *a*. While traversing the list, $curr_A$ and all nodes between $curr_A$ and *a* (including *a*) might be removed (denoted by a lighter node color). In such a case, thread *A* would proceed to the point where $curr_A$ points to *a*, and, without validation, would successfully remove *a*, even though it is no longer in the list. Validation is required to determine that *a* is no longer reachable from `head`.

The `OptimisticList` algorithm is not starvation-free, even if all node locks are individually starvation-free. A thread might be delayed forever if new nodes are repeatedly added and removed (see Exercise 107). Nevertheless, we would expect this algorithm to do well in practice, since starvation is rare.

9.7 Lazy Synchronization

The `OptimisticList` implementation works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking. One drawback of this particular algorithm is that `contains()` acquires locks, which is unattractive since `contains()` calls are likely to be much more common than calls to other methods.

The next step is to refine this algorithm so that `contains()` calls are wait-free, and `add()` and `remove()` methods, while still blocking, traverse the list only once (in the absence of contention). We add to each node a Boolean `marked` field indicating whether that node is in the set. Now, traversals do not need to lock the target node, and there is no need to validate that the node is reachable by retraversing the whole list. Instead, the algorithm maintains the invariant that every unmarked node is reachable. If a traversing thread does not find a node, or finds it marked, then that item is not in the set. As a result, `contains()` needs only one wait-free traversal. To add an element to the list, `add()` traverses the list, locks the target's predecessor, and inserts the node. The `remove()` method is lazy, taking two steps: first, mark the target node, *logically* removing it, and second, redirect its predecessor's `next` field, *physically* removing it.

In more detail, all methods traverse the list (possibly traversing logically and physically removed nodes) ignoring the locks. The `add()` and `remove()` methods lock the $\texttt{pred}_A$ and $\texttt{curr}_A$ nodes as before (Figs. 9.17 and 9.18), but validation does not retraverse the entire list (Fig. 9.16) to determine whether a node is in the set. Instead, because a node must be marked before being physically removed, validation need only check that $\texttt{curr}_A$ has not been marked. However, as Fig. 9.20 shows, for insertion and deletion, since $\texttt{pred}_A$ is the one being modified, one must also check that $\texttt{pred}_A$ itself is not marked, and that it points to $\texttt{curr}_A$. Logical removals require a small change to the abstraction map: an item is in the set if, and only if it is referred to by an *unmarked* reachable node. Notice

```
1    private boolean validate(Node pred, Node curr) {
2      return !pred.marked && !curr.marked && pred.next == curr;
3    }
```

Figure 9.16 The `LazyList` class: validation checks that neither the `pred` nor the `curr` node has been logically deleted, and that `pred` points to `curr`.

```
1    public boolean add(T item) {
2     int key = item.hashCode();
3     while (true) {
4        Node pred = head;
5        Node curr = head.next;
6        while (curr.key < key) {
7         pred = curr; curr = curr.next;
8        }
9        pred.lock();
10       try {
11         curr.lock();
12         try {
13           if (validate(pred, curr)) {
14             if (curr.key == key) {
15               return false;
16             } else {
17               Node node = new Node(item);
18               node.next = curr;
19               pred.next = node;
20               return true;
21             }
22           }
23         } finally {
24           curr.unlock();
25         }
26       } finally {
27         pred.unlock();
28       }
29     }
30   }
```

Figure 9.17 The `LazyList` class: `add()` method.

```
1   public boolean remove(T item) {
2     int key = item.hashCode();
3     while (true) {
4      Node pred = head;
5      Node curr = head.next;
6      while (curr.key < key) {
7       pred = curr; curr = curr.next;
8       }
9      pred.lock();
10     try {
11       curr.lock();
12       try {
13         if (validate(pred, curr)) {
14           if (curr.key != key) {
15             return false;
16           } else {
17             curr.marked = true;
18             pred.next = curr.next;
19             return true;
20           }
21         }
22       } finally {
23         curr.unlock();
24       }
25     } finally {
26       pred.unlock();
27     }
28    }
29  }
```

Figure 9.18 The `LazyList` class: the `remove()` method removes nodes in two steps, logical and physical.

```
1   public boolean contains(T item) {
2     int key = item.hashCode();
3     Node curr = head;
4     while (curr.key < key)
5       curr = curr.next;
6     return curr.key == key && !curr.marked;
7   }
```

Figure 9.19 The `LazyList` class: the `contains()` method.

that the path along which the node is reachable may contain marked nodes. The reader should check that any unmarked reachable node remains reachable, even if its predecessor is logically or physically deleted. As in the `OptimisticList` algorithm, `add()` and `remove()` are not starvation-free, because list traversals may be arbitrarily delayed by ongoing modifications.

The `contains()` method (Fig. 9.19) traverses the list once ignoring locks and returns *true* if the node it was searching for is present and unmarked, and *false*

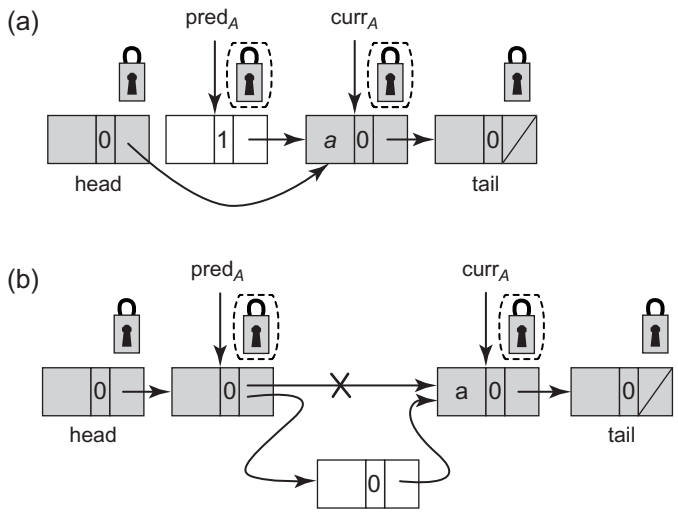

Figure 9.20 The `LazyList` class: why validation is needed. In Part (a) of the figure, thread *A* is attempting to remove node *a*. After it reaches the point where $\texttt{pred}_A$ refers to $\texttt{curr}_A$, and before it acquires locks on these nodes, the node $\texttt{pred}_A$ is logically and physically removed. After *A* acquires the locks, validation will detect the problem. In Part (b) of the figure, *A* is attempting to remove node *a*. After it reaches the point where $\texttt{pred}_A$ refers to $\texttt{curr}_A$, and before it acquires locks on these nodes, a new node is added between $\texttt{pred}_A$ and $\texttt{curr}_A$. After *A* acquires the locks, even though neither $\texttt{pred}_A$ or $\texttt{curr}_A$ are marked, validation detects that $\texttt{pred}_A$ is not the same as $\texttt{curr}_A$, and *A*'s call to `remove()` will be restarted.

otherwise. It is thus wait-free.[5] A marked node's value is ignored. Each time the traversal moves to a new node, the new node has a larger key than the previous one, even if the node is logically deleted.

The linearization points for `LazyList` `add()` and unsuccessful `remove()` calls are the same as for the `OptimisticList`. A successful `remove()` call is linearized when the mark is set (Line 17), and a successful `contains()` call is linearized when an unmarked matching node is found.

To understand how to linearize an unsuccessful `contains()`, let us consider the scenario depicted in Fig. 9.21. In Part (a), node *a* is marked as removed (its `marked` field is set) and thread *A* is attempting to find the node matching *a*'s key. While *A* is traversing the list, $\texttt{curr}_A$ and all nodes between $\texttt{curr}_A$ and *a* including *a* are removed, both logically and physically. Thread *A* would still proceed to the point where $\texttt{curr}_A$ points to *a*, and would detect that *a* is marked and no longer in the abstract set. The call could be linearized at this point.

Now let us consider the scenario depicted in Part (b). While *A* is traversing the removed section of the list leading to *a*, and before it reaches the removed

5 Notice that the list ahead of a given traversing thread cannot grow forever due to newly inserted keys, since the key size is finite.

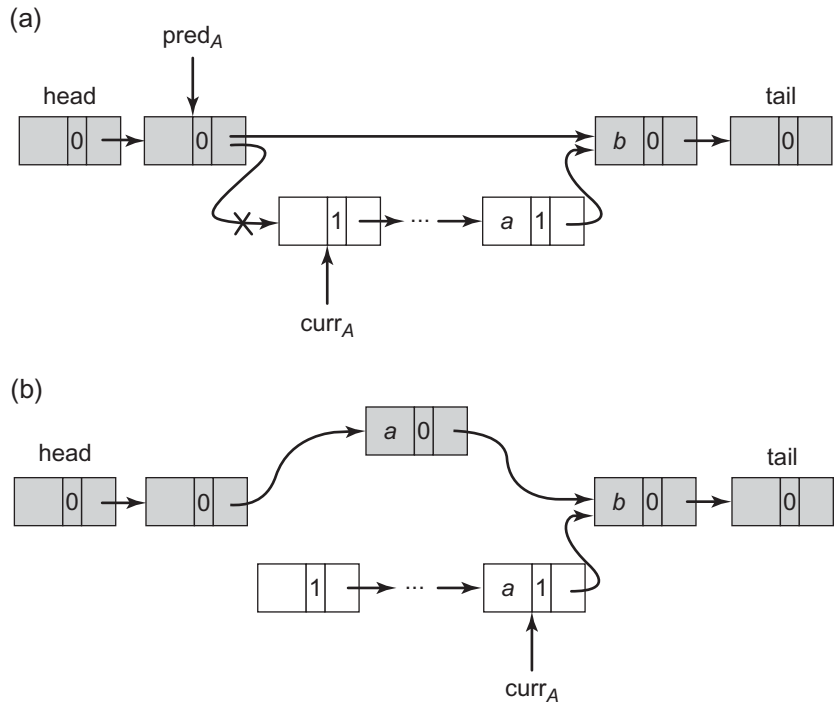

Figure 9.21 The `LazyList` class: linearizing an unsuccessful `contains()` call. Dark nodes are physically in the list and white nodes are physically removed. In Part (a), while thread *A* is traversing the list, a concurrent `remove()` call disconnects the sublist referred to by `curr`. Notice that nodes with items *a* and *b* are still reachable, so whether an item is actually in the list depends only on whether it is marked. Thread *A*'s call is linearized at the point when it sees that *a* is marked and is no longer in the abstract set. Alternatively, let us consider the scenario depicted in Part (b). While thread *A* is traversing the list leading to marked node *a*, another thread adds a new node with key *a*. It would be wrong to linearize thread *A*'s unsuccessful `contains()` call to when it found the marked node *a*, since this point occurs after the insertion of the new node with key *a* to the list.

node *a*, another thread adds a new node with a key *a* to the reachable part of the list. Linearizing thread *A*'s unsuccessful `contains()` method at the point it finds the marked node *a* would be wrong, since this point occurs *after* the insertion of the new node with key *a* to the list. We therefore linearize an unsuccessful `contains()` method call within its execution interval at the earlier of the following points: (1) the point where a removed matching node, or a node with a key greater than the one being searched for is found, and (2) the point immediately before a new matching node is added to the list. Notice that the second is guaranteed to be within the execution interval because the insertion of the new node with the same key must have happened after the start of the `contains()` method, or the `contains()` method would have found that item. As can be

seen, the linearization point of the unsuccessful `contains`() is determined by the ordering of events in the execution, and is not a predetermined point in the method's code.

One benefit of lazy synchronization is that we can separate unobtrusive logical steps such as setting a flag, from disruptive physical changes to the structure, such as disconnecting a node. The example presented here is simple because we disconnect one node at a time. In general, however, delayed operations can be batched and performed lazily at a convenient time, reducing the overall disruptiveness of physical modifications to the structure.

The principal disadvantage of the `LazyList` algorithm is that `add`() and `remove`() calls are blocking: if one thread is delayed, then others may also be delayed.

9.8 Non-Blocking Synchronization

We have seen that it is sometimes a good idea to mark nodes as logically removed before physically removing them from the list. We now show how to extend this idea to eliminate locks altogether, allowing all three methods, `add`(), `remove`(), and `contains`(), to be nonblocking. (The first two methods are lock-free and the last wait-free). A naïve approach would be to use `compareAndSet()` to change the `next` fields. Unfortunately, this idea does not work. In Fig. 9.22, part (a) shows a thread *A* attempting to remove a node *a* while thread *B* is adding a node *b*. Suppose *A* applies `compareAndSet()` to `head.next`, while *B* applies `compareAndSet()` to `a.next`. The net effect is that *a* is correctly deleted but *b* is not added to the list. In part (b) of the figure, *A* attempts to remove *a*, the first node in the list, while *B* is about to remove *b*, where a points to *b*. Suppose *A* applies `compareAndSet()` to `head.next`, while *B* applies `compareAndSet()` to `a.next`. The net effect is to remove *a*, but not *b*.

If *B* wants to remove $\texttt{curr}_B$ from the list, it might call `compareAndSet()` to set $\texttt{pred}_B$'s next field to $\texttt{curr}_B$'s successor. It is not hard to see that if these two threads try to remove these adjacent nodes concurrently, the list will end up with b not being removed. A similar situation for a pair of concurrent `add`() and `remove`() methods is depicted in the upper part of Fig. 9.22.

Clearly, we need a way to ensure that a node's fields cannot be updated, after that node has been logically or physically removed from the list. Our approach is to treat the node's `next` and `marked` fields as a single atomic unit: any attempt to update the `next` field when the `marked` field is *true* will fail.

Pragma 9.8.1. An `AtomicMarkableReference<T>` is an object from the java.util.concurrent.atomic package that encapsulates both a reference to an object of type `T` and a Boolean `mark`. These fields can be updated atomically, either together or individually. For example, the `compareAndSet()` method

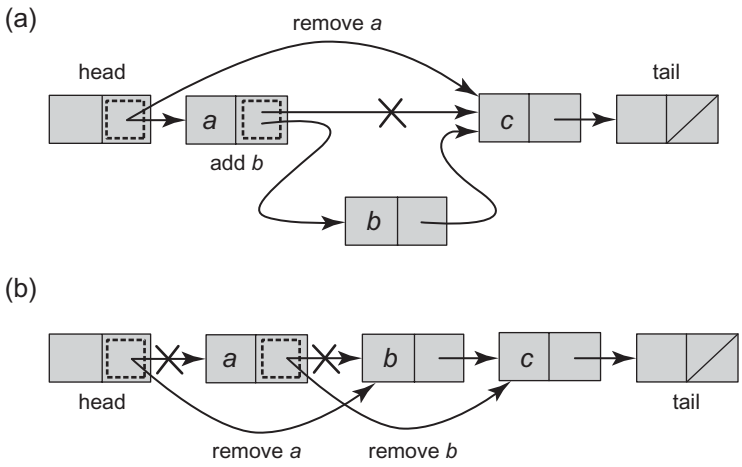

Figure 9.22 The `LockFreeList` class: why mark and reference fields must be modified atomically. In Part (a) of the figure, thread *A* is about to remove *a*, the first node in the list, while *B* is about to add *b*. Suppose *A* applies `compareAndSet()` to `head.next`, while *B* applies `compareAndSet()` to `a.next`. The net effect is that *a* is correctly deleted but *b* is not added to the list. In Part (b) of the figure, thread *A* is about to remove *a*, the first node in the list, while *B* is about to remove *b*, where *a* points to *b*. Suppose *A* applies `compareAndSet()` to `head.next`, while *B* applies `compareAndSet()` to `a.next`. The net effect is to remove *a*, but not *b*.

tests the expected reference and mark values, and if both tests succeed, replaces them with updated reference and mark values. As shorthand, the `attemptMark()` method tests an expected reference value and if the test succeeds, replaces it with a new mark value. The `get()` method has an unusual interface: it returns the object's reference value and stores the mark value in a Boolean array argument. Fig. 9.23 illustrates the interfaces of these methods.

```
1  public boolean compareAndSet(T expectedReference,
2                               T newReference,
3                               boolean expectedMark,
4                               boolean newMark);
5  public boolean attemptMark(T expectedReference,
6                             boolean newMark);
7  public T get(boolean[] marked);
```

Figure 9.23 Some `AtomicMarkableReference<T>` methods: the `compareAndSet()` method tests and updates both the mark and reference fields, while the `attemptMark()` method updates the mark if the reference field has the expected value. The `get()` method returns the encapsulated reference and stores the mark at position 0 in the argument array.

In C or C++, one could provide this functionality efficiently by "stealing" a bit from a pointer, using bit-wise operators to extract the mark and the pointer from a single word. In Java, of course, one cannot manipulate pointers directly, so this functionality must be provided by a library.

As described in detail in Pragma 9.8.1, an `AtomicMarkableReference<T>` object encapsulates both a reference to an object of type `T` and a Boolean `mark`. These fields can be atomically updated, either together or individually.

We make each node's `next` field an `AtomicMarkableReference<Node>`. Thread A logically removes $\texttt{curr}_A$ by setting the mark bit in the node's `next` field, and shares the physical removal with other threads performing `add()` or `remove()`: as each thread traverses the list, it cleans up the list by physically removing (using `compareAndSet()`) any marked nodes it encounters. In other words, threads performing `add()` and `remove()` do not traverse marked nodes, they remove them before continuing. The `contains()` method remains the same as in the `LazyList` algorithm, traversing all nodes whether they are marked or not, and testing if an item is in the list based on its key and mark.

It is worth pausing to consider a design decision that differentiates the `LockFreeList` algorithm from the `LazyList` algorithm. Why do threads that add or remove nodes never traverse marked nodes, and instead physically remove all marked nodes they encounter? Suppose that thread A were to traverse marked nodes without physically removing them, and after logically removing $\texttt{curr}_A$, were to attempt to physically remove it as well. It could do so by calling `compareAndSet()` to try to redirect $\texttt{pred}_A$'s `next` field, simultaneously verifying that $\texttt{pred}_A$ is not marked and that it refers to $\texttt{curr}_A$. The difficulty is that because A is not holding locks on $\texttt{pred}_A$ and $\texttt{curr}_A$, other threads could insert new nodes or remove $\texttt{pred}_A$ before the `compareAndSet()` call.

Consider a scenario in which another thread marks $\texttt{pred}_A$. As illustrated in Fig. 9.22, we cannot safely redirect the `next` field of a marked node, so A would have to restart the physical removal by retraversing the list. This time, however, A would have to physically remove $\texttt{pred}_A$ before it could remove $\texttt{curr}_A$. Even worse, if there is a sequence of logically removed nodes leading to $\texttt{pred}_A$, A must remove them all, one after the other, before it can remove $\texttt{curr}_A$ itself.

This example illustrates why `add()` and `remove()` calls do not traverse marked nodes: when they arrive at the node to be modified, they may be forced to retraverse the list to remove previous marked nodes. Instead, we choose to have both `add()` and `remove()` physically remove any marked nodes on the path to their target node. The `contains()` method, by contrast, performs no modification, and therefore need not participate in the cleanup of logically removed nodes, allowing it, as in the `LazyList`, to traverse both marked and unmarked nodes.

In presenting our LockFreeList algorithm, we factor out functionality common to the add() and remove() methods by creating an inner Window class to help navigation. As shown in Fig. 9.24, a Window object is a structure with pred and curr fields. The find() method takes a head node and a key *a*, and traverses the list, seeking to set pred to the node with the largest key less than *a*, and curr to the node with the least key greater than or equal to *a*. As thread *A* traverses the list, each time it advances $curr_A$, it checks whether that node is marked (Line 16). If so, it calls compareAndSet() to attempt to physically remove the node by setting $pred_A$'s next field to $curr_A$'s next field. This call tests both the field's reference and Boolean mark values, and fails if either value has changed. A concurrent thread could change the mark value by logically removing $pred_A$, or it could change the reference value by physically removing $curr_A$. If the call fails, *A* restarts the traversal from the head of the list; otherwise the traversal continues.

The LockFreeList algorithm uses the same abstraction map as the LazyList algorithm: an item is in the set if, and only if it is in an *unmarked* reachable node.

```
1   class Window {
2     public Node pred, curr;
3     Window(Node myPred, Node myCurr) {
4       pred = myPred; curr = myCurr;
5     }
6   }
7   public Window find(Node head, int key) {
8     Node pred = null, curr = null, succ = null;
9     boolean[] marked = {false};
10    boolean snip;
11    retry: while (true) {
12      pred = head;
13      curr = pred.next.getReference();
14      while (true) {
15        succ = curr.next.get(marked);
16        while (marked[0]) {
17          snip = pred.next.compareAndSet(curr, succ, false, false);
18          if (!snip) continue retry;
19          curr = succ;
20          succ = curr.next.get(marked);
21        }
22        if (curr.key >= key)
23          return new Window(pred, curr);
24        pred = curr;
25        curr = succ;
26      }
27    }
28  }
```

Figure 9.24 The Window class: the find() method returns a structure containing the nodes on either side of the key. It removes marked nodes when it encounters them.

The compareAndSet() call at Line 17 of the find() method is an example of a *benevolent side effect*: it changes the concrete list without changing the abstract set, because removing a marked node does not change the value of the abstraction map.

Fig. 9.25 shows the LockFreeList class's add() method. Suppose thread A calls add(a). A uses find() to locate pred_A and curr_A. If curr_A's key is equal to a's, the call returns *false*. Otherwise, add() initializes a new node a to hold a, and sets a to refer to curr_A. It then calls compareAndSet() (Line 11) to set pred_A to a. Because the compareAndSet() tests both the mark and the reference, it succeeds only if pred_A is unmarked and refers to curr_A. If the compareAndSet() is successful, the method returns *true*, and otherwise it starts over.

Fig. 9.26 shows the LockFreeList algorithm's remove() method. When A calls remove() to remove item a, it uses find() to locate pred_A and curr_A. If curr_A's key fails to match a's, the call returns *false*. Otherwise, remove() uses a compareAndSet() to attempt to mark curr_A as logically removed (Line 27). This call succeeds only if no other thread has set the mark first. If it succeeds, the call returns *true*. A single attempt is made to physically remove the node, but there is no need to try again because the node will be removed by the next thread to traverse that region of the list. If the compareAndSet() call fails, remove() starts over.

The LockFreeList algorithm's contains() method is virtually the same as that of the LazyList (Fig. 9.27). There is one small change: to test if curr is marked we must apply curr.next.get(marked) and check that marked[0] is *true*.

```
1  public boolean add(T item) {
2    int key = item.hashCode();
3    while (true) {
4     Window window = find(head, key);
5     Node pred = window.pred, curr = window.curr;
6     if (curr.key == key) {
7         return false;
8       } else {
9         Node node = new Node(item);
10        node.next = new AtomicMarkableReference(curr, false);
11        if (pred.next.compareAndSet(curr, node, false, false)) {
12          return true;
13        }
14      }
15    }
16  }
```

Figure 9.25 The LockFreeList class: the add() method calls find() to locate pred_A and curr_A. It adds a new node only if pred_A is unmarked and refers to curr_A.

```
17    public boolean remove(T item) {
18      int key = item.hashCode();
19      boolean snip;
20      while (true) {
21       Window window = find(head, key);
22       Node pred = window.pred, curr = window.curr;
23       if (curr.key != key) {
24         return false;
25       } else {
26         Node succ = curr.next.getReference();
27         snip = curr.next.compareAndSet(succ, succ, false, true);
28         if (!snip)
29           continue;
30         pred.next.compareAndSet(curr, succ, false, false);
31         return true;
32       }
33     }
34   }
```

Figure 9.26 The `LockFreeList` class: the `remove()` method calls `find()` to locate $pred_A$ and $curr_A$, and atomically marks the node for removal.

```
35    public boolean contains(T item) {
36      boolean[] marked = false;
37      int key = item.hashCode();
38      Node curr = head;
39      while (curr.key < key) {
40       curr = curr.next.getReference();
41       Node succ = curr.next.get(marked);
42      }
43      return (curr.key == key && !marked[0])
44    }
```

Figure 9.27 The `LockFreeList` class: the wait-free `contains()` method is almost the same as in the `LazyList` class. There is one small difference: it calls `curr.next.get(marked)` to test whether `curr` is marked.

9.9 Discussion

We have seen a progression of list-based lock implementations in which the granularity and frequency of locking was gradually reduced, eventually reaching a fully nonblocking list. The final transition from the `LazyList` to the `LockFreeList` exposes some of the design decisions that face concurrent programmers. As we will see, approaches such as optimistic and lazy synchronization will appear time and again when designing more complex data structures.

On the one hand, the `LockFreeList` algorithm guarantees progress in the face of arbitrary delays. However, there is a price for this strong progress guarantee:

- The need to support atomic modification of a reference and a Boolean mark has an added performance cost.[6]
- As `add()` and `remove()` traverse the list, they must engage in concurrent cleanup of removed nodes, introducing the possibility of contention among threads, sometimes forcing threads to restart traversals, even if there was no change near the node each was trying to modify.

On the other hand, the lazy lock-based list does not guarantee progress in the face of arbitrary delays: its `add()` and `remove()` methods are blocking. However, unlike the lock-free algorithm, it does not require each node to include an atomically markable reference. It also does not require traversals to clean up logically removed nodes; they progress down the list, ignoring marked nodes.

Which approach is preferable depends on the application. In the end, the balance of factors such as the potential for arbitrary thread delays, the relative frequency of calls to the `add()` and `remove()` methods, the overhead of implementing an atomically markable reference, and so on determine the choice of whether to lock, and if so, at what granularity.

9.10 Chapter Notes

Lock coupling was invented by Rudolf Bayer and Mario Schkolnick [19]. The first designs of lock-free linked-list algorithms are credited to John Valois [147]. The Lock-free list implementation shown here is a variation on the lists of Maged Michael [114], who based his work on earlier linked-list algorithms by Tim Harris [53]. This algorithm is referred to by many as the Harris-Michael algorithm. The Harris-Michael algorithm is the one used in the Java Concurrency Package. The `OptimisticList` algorithm was invented for this chapter, and the lazy algorithm is credited to Steven Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Nir Shavit, and Bill Scherer [55].

9.11 Exercises

Exercise 100. Describe how to modify each of the linked list algorithms if object hash codes are not guaranteed to be unique.

---

6 In the Java Concurrency Package, for example, this cost is somewhat reduced by using a reference to an intermediate dummy node to signify that the `marked` bit is set.

Exercise 101. Explain why the fine-grained locking algorithm is not subject to deadlock.

Exercise 102. Explain why the fine-grained list's `add()` method is linearizable.

Exercise 103. Explain why the optimistic and lazy locking algorithms are not subject to deadlock.

Exercise 104. Show a scenario in the optimistic algorithm where a thread is forever attempting to delete a node.

Hint: since we assume that all the individual node locks are starvation-free, the livelock is not on any individual lock, and a bad execution must repeatedly add and remove nodes from the list.

Exercise 105. Provide the code for the `contains()` method missing from the fine-grained algorithm. Explain why your implementation is correct.

Exercise 106. Is the optimistic list implementation still correct if we switch the order in which `add()` locks the `pred` and `curr` entries?

Exercise 107. Show that in the optimistic list algorithm, if $\texttt{pred}_A$ is not *null*, then `tail` is reachable from $\texttt{pred}_A$, even if $\texttt{pred}_A$ itself is not reachable.

Exercise 108. Show that in the optimistic algorithm, the `add()` method needs to lock only `pred`.

Exercise 109. In the optimistic algorithm, the `contains()` method locks two entries before deciding whether a key is present. Suppose, instead, it locks no entries, returning *true* if it observes the value, and *false* otherwise.

Either explain why this alternative is linearizable, or give a counterexample showing it is not.

Exercise 110. Would the lazy algorithm still work if we marked a node as removed simply by setting its `next` field to *null*? Why or why not? What about the lock-free algorithm?

Exercise 111. In the lazy algorithm, can $\texttt{pred}_A$ ever be unreachable? Justify your answer.

Exercise 112. Your new employee claims that the lazy list's validation method (Fig. 9.16) can be simplified by dropping the check that `pred.next` is equal to `curr`. After all, the code always sets `pred` to the old value of `curr`, and before `pred.next` can be changed, the new value of `curr` must be marked, causing the validation to fail. Explain the error in this reasoning.

Exercise 113. Can you modify the lazy algorithm's `remove()` so it locks only one node?

Exercise 114. In the lock-free algorithm, argue the benefits and drawbacks of having the `contains`() method help in the cleanup of logically removed entries.

Exercise 115. In the lock-free algorithm, if an `add`() method call fails because `pred` does not point to `curr`, but `pred` is not marked, do we need to traverse the list again from `head` in order to attempt to complete the call?

Exercise 116. Would the `contains`() method of the lazy and lock-free algorithms still be correct if logically removed entries were not guaranteed to be sorted?

Exercise 117. The `add`() method of the lock-free algorithm never finds a marked node with the same key. Can one modify the algorithm so that it will simply insert its new added object into the existing marked node with same key if such a node exists in the list, thus saving the need to insert a new node?

Exercise 118. Explain why the following cannot happen in the `LockFreeList` algorithm. A node with item x is logically but not yet physically removed by some thread, then the same item x is added into the list by another thread, and finally a `contains`() call by a third thread traverses the list, finding the logically removed node, and returning *false*, even though the linearization order of the `remove`() and `add`() implies that x is in the set.

This page intentionally left blank

10 Concurrent Queues and the ABA Problem

10.1 Introduction

In the subsequent chapters, we look at a broad class of objects known as *pools*. A pool is similar to the `Set` class studied in Chapter 9, with two main differences: a pool does not necessarily provide a `contains()` method to test membership, and it allows the same item to appear more than once. The `Pool` has `get()` and `set()` methods as in Fig. 10.1. Pools show up in many places in concurrent systems. For example, in many applications, one or more *producer* threads produce items to be consumed by one or more *consumer* threads. These items may be jobs to perform, keystrokes to interpret, purchase orders to execute, or packets to decode. Sometimes, producers are *bursty*, suddenly and briefly producing items faster than consumers can consume them. To allow consumers to keep up, we can place a *buffer* between the producers and the consumers. Items produced faster than they can be consumed accumulate in the buffer, from which they are consumed as quickly as possible. Often, pools act as producer–consumer buffers.

Pools come in several varieties.

- A pool can be *bounded* or *unbounded*. A bounded pool holds a limited number of items. This limit is called its *capacity*. By contrast, an unbounded pool can hold any number of items. Bounded pools are useful when we want to keep producer and consumer threads loosely synchronized, ensuring that producers do not get too far ahead of consumers. Bounded pools may also be simpler to implement than unbounded pools. On the other hand, unbounded pools are useful when there is no need to set a fixed limit on how far producers can outstrip consumers.
- Pool methods may be *total*, *partial*, or *synchronous*.
  - A method is *total* if calls do not wait for certain conditions to become true. For example, a `get()` call that tries to remove an item from an empty pool immediately returns a failure code or throws an exception. If the pool is bounded, a total `set()` that tries to add an item to a full pool immediately

The Art of Multiprocessor Programming. DOI: 10.1016/B978-0-12-397337-5.00010-1

```
1  public interface Pool<T> {
2    void set(T item);
3    T get();
4  }
```

Figure 10.1 The `Pool<T>` interface.

returns a failure code or an exception. A total interface makes sense when the producer (or consumer) thread has something better to do than wait for the method call to take effect.

— A method is *partial* if calls may wait for conditions to hold. For example, a partial `get()` call that tries to remove an item from an empty pool blocks until an item is available to return. If the pool is bounded, a partial `set()` call that tries to add an item to a full pool blocks until an empty slot is available to fill. A partial interface makes sense when the producer (or consumer) has nothing better to do than to wait for the pool to become nonfull (or nonempty).

— A method is *synchronous* if it waits for another method to overlap its call interval. For example, in a synchronous pool, a method call that adds an item to the pool is blocked until that item is removed by another method call. Symmetrically, a method call that removes an item from the pool is blocked until another method call makes an item available to be removed. (Such methods are partial.) Synchronous pools are used for communication in programming languages such as CSP and Ada in which threads *rendezvous* to exchange information.

- Pools provide different *fairness* guarantees. They can be first-in-first-out (a queue), last-in-first-out (a stack), or other, weaker properties. The importance of fairness when buffering using a pool is clear to anyone who has ever called a bank or a technical support line, only to be placed in a pool of waiting calls. The longer you wait, the more consolation you draw from the recorded message asserting that calls are answered in the order they arrive. Perhaps.

10.2 Queues

In this chapter we consider a kind of pool that provides *first-in-first-out* (FIFO) fairness. A sequential `Queue<T>` is an ordered sequence of items (of type `T`). It provides an `enq(`x`)` method that puts item x at one end of the queue, called the *tail*, and a `deq()` method that removes and returns the item at the other end of the queue, called the *head*. A concurrent queue is linearizable to a sequential queue. Queues are pools, where `enq()` implements `set()`, and `deq()` implements `get()`. We use queue implementations to illustrate a number of important principles. In later chapters we consider pools that provide other fairness guarantees.

10.3 A Bounded Partial Queue

For simplicity, we assume it is illegal to add a *null* value to a queue. Of course, there may be circumstances where it makes sense to add and remove *null* values, but we leave it as an exercise to adapt our algorithms to accommodate *null* values.

How much concurrency can we expect a bounded queue implementation with multiple concurrent enqueuers and dequeuers to provide? Very informally, the `enq()` and `deq()` methods operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an `enq()` call and a `deq()` call should, in principle, be able to proceed without interference. For the same reason, concurrent `enq()` calls probably will interfere, and the same holds for `deq()` calls. This informal reasoning may sound convincing, and it is in fact mostly correct, but realizing this level of concurrency is not trivial.

Here, we implement a bounded queue as a linked list. (We could also have used an array.) Fig. 10.2 shows the queue's fields and constructor, Figs. 10.3 and 10.4 show the `enq()` and `deq()` methods, and Fig. 10.5 shows a queue node. Like the lists studied in Chapter 9, a queue node has `value` and `next` fields.

As seen in Fig. 10.6, the queue itself has `head` and `tail` fields that respectively refer to the first and last nodes in the queue. The queue always contains a *sentinel* node acting as a place-holder. Like the sentinel nodes in Chapter 9, it marks a position in the queue, though its value is meaningless. Unlike the list algorithms in Chapter 9, in which the same nodes always act as sentinels, the queue repeatedly replaces the sentinel node. We use two distinct locks, `enqLock` and `deqLock`, to ensure that at most one enqueuer, and at most one dequeuer at a time can manipulate the queue object's fields. Using two locks instead of one ensures that an enqueuer does not lock out a dequeuer unnecessarily, and vice versa. Each lock has an associated *condition* field. The `enqLock` is associated with

```
1  public class BoundedQueue<T> {
2    ReentrantLock enqLock, deqLock;
3    Condition notEmptyCondition, notFullCondition;
4    AtomicInteger size;
5    volatile Node head, tail;
6    int capacity;
7    public BoundedQueue(int _capacity) {
8      capacity = _capacity;
9      head = new Node(null);
10     tail = head;
11     size = new AtomicInteger(0);
12     enqLock = new ReentrantLock();
13     notFullCondition = enqLock.newCondition();
14     deqLock = new ReentrantLock();
15     notEmptyCondition = deqLock.newCondition();
16   }
```

Figure 10.2 The BoundedQueue class: fields and constructor.

```
17    public void enq(T x) {
18      boolean mustWakeDequeuers = false;
19      enqLock.lock();
20      try {
21        while (size.get() == capacity)
22          notFullCondition.await();
23        Node e = new Node(x);
24        tail.next = tail; tail = e;
25        if (size.getAndIncrement() == 0)
26          mustWakeDequeuers = true;
27      } finally {
28        enqLock.unlock();
29      }
30      if (mustWakeDequeuers) {
31        deqLock.lock();
32        try {
33          notEmptyCondition.signalAll();
34        } finally {
35          deqLock.unlock();
36        }
37      }
38    }
```

Figure 10.3 The BoundedQueue class: the enq() method.

```
39    public T deq() {
40      T result;
41      boolean mustWakeEnqueuers = false;
42      deqLock.lock();
43      try {
44        while (size.get() == 0)
45          notEmptyCondition.await();
46        result = head.next.value;
47        head = head.next;
48        if (size.getAndDecrement() == capacity) {
49          mustWakeEnqueuers = true;
50        }
51      } finally {
52        deqLock.unlock();
53      }
54      if (mustWakeEnqueuers) {
55        enqLock.lock();
56        try {
57          notFullCondition.signalAll();
58        } finally {
59          enqLock.unlock();
60        }
61      }
62      return result;
63    }
```

Figure 10.4 The BoundedQueue class: the deq() method.

```
64    protected class Node {
65      public T value;
66      public volatile Node next;
67      public Node(T x) {
68        value = x;
69        next = null;
70      }
71    }
72  }
```

Figure 10.5 BoundedQueue class: List Node.

the notFullCondition condition, used to notify waiting dequeuers when the queue is no longer full. The deqLock is associated with notEmptyCondition, used to notify waiting enqueuers when the queue is no longer empty.

Since the queue is bounded, we must keep track of the number of empty slots. The size field is an AtomicInteger that tracks the number of objects currently in the queue. This field is decremented by deq() calls and incremented by enq() calls.

The enq() method (Fig. 10.3) works as follows. A thread acquires the enqLock (Line 19), and reads the size field (Line 21). While that field is equal to the capacity, the queue is full, and the enqueuer must wait until a dequeuer makes room. The enqueuer waits on the notFullCondition field (Line 22), releasing the enqueue lock temporarily, and blocking until that condition is signaled. Each time the thread awakens (Line 22), it checks whether there is room, and if not, goes back to sleep.

Once the number of empty slots exceeds zero, however, the enqueuer may proceed. We note that once the enqueuer observes an empty slot, while the enqueue is in progress no other thread can fill the queue, because all the other enqueuers are locked out, and a concurrent dequeuer can only increase the number of empty slots. (Synchronization for the enq() method is symmetric.)

We must carefully check that this implementation does not suffer from the kind of "lost-wakeup" bug described in Chapter 8. Care is needed because an enqueuer encounters a full queue in two steps: first, it sees that size is the queue capacity, and second, it waits on notFullCondition until there is room in the queue. When a dequeuer changes the queue from full to not-full, it acquires enqLock and signals notFullCondition. Even though the size field is not protected by the enqLock, the dequeuer acquires the enqLock before it signals the condition, so the dequeuer cannot signal between the enqueuer's two steps.

The deq() method proceeds as follows. It reads the size field to check whether the queue is empty. If so, the dequeuer must wait until an item is enqueued. Like the enq() method, the dequeuer waits on notEmptyCondition, which temporarily releases deqLock, and blocks until the condition is signaled. Each time the thread awakens, it checks whether the queue is empty, and if so, goes back to sleep.

It is important to understand that the abstract queue's head and tail items are not always the same as those referenced by head and tail. An item is logically added to the queue as soon as the last node's next field is redirected to the

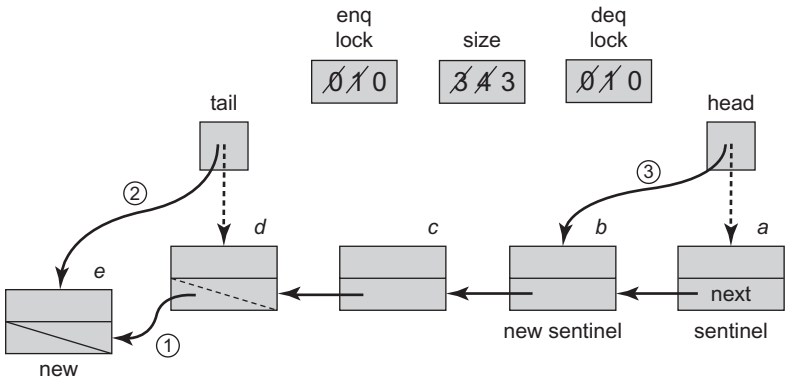

Figure 10.6 The `enq()` and `deq()` methods of the `BoundedQueue` with 4 slots. First a node is enqueued into the queue by acquiring the `enqLock`. The `enq()` checks that the `size` is 3 which is less than the bound. It then redirects the `next` field of the node referenced by the `tail` field (step 1), redirects `tail` to the new node (step 2), increments the `size` to 4, and releases the lock. Since `size` is now 4, any further calls to `enq()` will cause the threads to block until the `notFullCondition` is signalled by some `deq()`. Next, a node is dequeued from the queue by some thread. The `deq()` acquires the `deqLock`, reads the new value *b* from the successor of the node referenced by `head` (this node is the current sentinel), redirects `head` to this successor node (step 3), decrements the `size` to 3, and releases the lock. Before completing the `deq()`, because the `size` was 4 when it started, the thread acquires the `enqLock` and signals any enqueuers waiting on `notFullCondition` that they can proceed.

new item (the linearization point of the `enq()`), even if the enqueuer has not yet updated `tail`. For example, a thread can hold the `enqLock` and be in the process of inserting a new node. Suppose it has not yet redirected the `tail` field. A concurrent dequeuing thread could acquire the `deqLock`, read and return the new node's value, and redirect the `head` to the new node, all before the enqueuer redirects `tail` to the newly inserted node.

Once the dequeuer establishes that the queue is nonempty, the queue will remain nonempty for the duration of the `deq()` call, because all other dequeuers have been locked out. Consider the first nonsentinel node in the queue (i.e., the node referenced by the sentinel node's `next` field). The dequeuer reads this node's `value` field, and sets the queue's `head` to refer to it, making it the new sentinel node. The dequeuer then decrements size and releases deqLock. If the dequeuer found the former size was the queue capacity, then there may be enqueuers waiting on `notEmptyCondition`, so the dequeuer acquires `enqLock`, and signals all such threads to wake up.

One drawback of this implementation is that concurrent `enq()` and `deq()` calls interfere with each other, but not through locks. All method calls apply `getAndIncrement()` or `getAndDecrement()` calls to the `size` field. These methods are more expensive than ordinary reads–writes, and they could cause a sequential bottleneck.

One way to reduce such interactions is to split this field into two counters: `enqSideSize` is an integer field incremented by `enq()`, and `deqSideSize` is an integer field decremented by `deq()`. A thread calling `enq()` tests `enqSideSize`, and as long as it is less than the capacity, it proceeds. When the field reaches capacity, the thread locks `deqLock`, adds `deqSize` to `EnqSize`, and resets `deqSideSize` to 0. Instead of synchronizing on every method call, this technique synchronizes sporadically when the enqueuer's size estimate becomes too large.

10.4 An Unbounded Total Queue

We now describe a different kind of queue that can hold an unbounded number of items. The `enq()` method always enqueues its item, and `deq()` throws `EmptyException` if there is no item to dequeue. The representation is the same as the bounded queue, except there is no need to count the number of items in the queue, or to provide conditions on which to wait. As illustrated in Figs. 10.7 and 10.8, this algorithm is simpler than the bounded algorithm.

```
1   public void enq(T x) {
2     enqLock.lock();
3     try {
4       Node e = new Node(x);
5       tail.next = e;
6       tail = e;
7     } finally {
8       enqLock.unlock();
9     }
10  }
```

Figure 10.7 The `UnboundedQueue<T>` class: the `enq()` method.

```
11  public T deq() throws EmptyException {
12    T result;
13    deqLock.lock();
14    try {
15      if (head.next == null) {
16        throw new EmptyException();
17      }
18      result = head.next.value;
19      head = head.next;
20    } finally {
21      deqLock.unlock();
22    }
23    return result;
24  }
```

Figure 10.8 The `UnboundedQueue<T>` class: the `deq()` method.

This queue cannot deadlock, because each method acquires only one lock, either `enqLock` or `deqLock`. A sentinel node alone in the queue will never be deleted, so each `enq()` call will succeed as soon as it acquires the lock. Of course, a `deq()` method may fail if the queue is empty (i.e., if `head.next` is *null*). As in the earlier queue implementations, an item is actually enqueued when the `enq()` call sets the last node's `next` field to the new node, even before `enq()` resets `tail` to refer to the new node. After that instant, the new item is reachable along a chain of the `next` references. As usual, the queue's actual head and tail are not necessarily the items referenced by `head` and `tail`. Instead, the actual head is the successor of the node referenced by `head`, and the actual tail is the last item reachable from the head. Both the `enq()` and `deq()` methods are total as they do not wait for the queue to become empty or full.

10.5 An Unbounded Lock-Free Queue

We now describe the `LockFreeQueue<T>` class, an unbounded lock-free queue implementation. This class, depicted in Figs. 10.9 through 10.11, is a natural

```
1  public class Node {
2      public T value;
3      public AtomicReference<Node> next;
4      public Node(T value) {
5        this.value = value;
6        next = new AtomicReference<Node>(null);
7      }
8    }
```

Figure 10.9 The `LockFreeQueue<T>` class: list node.

```
 9    public void enq(T value) {
10      Node node = new Node(value);
11      while (true) {
12        Node last = tail.get();
13        Node next = last.next.get();
14        if (last == tail.get()) {
15          if (next == null) {
16            if (last.next.compareAndSet(next, node)) {
17              tail.compareAndSet(last, node);
18              return;
19            }
20          } else {
21            tail.compareAndSet(last, next);
22          }
23        }
24      }
25    }
```

Figure 10.10 The `LockFreeQueue<T>` class: the `enq()` method.

```
26    public T deq() throws EmptyException {
27      while (true) {
28        Node first = head.get();
29        Node last = tail.get();
30        Node next = first.next.get();
31        if (first == head.get()) {
32          if (first == last) {
33            if (next == null) {
34              throw new EmptyException();
35            }
36            tail.compareAndSet(last, next);
37          } else {
38            T value = next.value;
39            if (head.compareAndSet(first, next))
40              return value;
41          }
42        }
43      }
44    }
```

Figure 10.11 The LockFreeQueue<T> class: the deq() method.

extension of the unbounded total queue of Section 10.4. Its implementation prevents method calls from starving by having the quicker threads help the slower threads.

As done earlier, we represent the queue as a list of nodes. However, as shown in Fig. 10.9, each node's `next` field is an `AtomicReference<Node>` that refers to the next node in the list. As can be seen in Fig. 10.12, the queue itself consists of two `AtomicReference<Node>` fields: `head` refers to the first node in the queue, and `tail` to the last. Again, the first node in the queue is a sentinel node, whose value is meaningless. The queue constructor sets both `head` and `tail` to refer to the sentinel.

An interesting aspect of the `enq()` method is that it is *lazy*: it takes place in two distinct steps. To make this method lock-free, threads may need to help one another. Fig. 10.12 illustrates these steps.

In the following description the line numbers refer to Figs. 10.9 through 10.11. Normally, the `enq()` method creates a new node (Line 10), locates the last node in the queue (Lines 12–13), and performs the following two steps:

1. It calls `compareAndSet()` to append the new node (Line 16), and
2. calls `compareAndSet()` to change the queue's `tail` field from the prior last node to the current last node (Line 17).

Because these two steps are not executed atomically, every other method call must be prepared to encounter a half-finished `enq()` call, and to finish the job. This is a real-world example of the "helping" technique we first saw in the universal construction of Chapter 6.

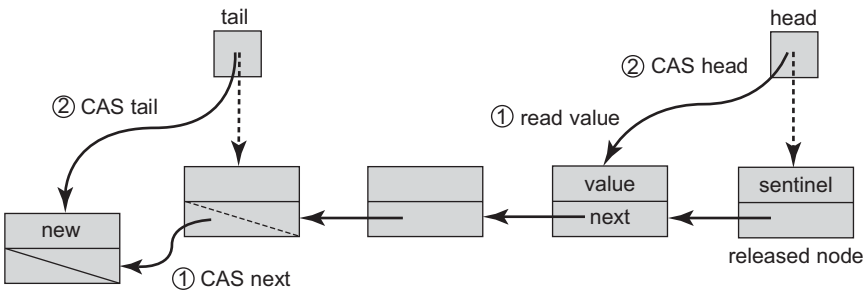

Figure 10.12 The lazy lock-free `enq()` and `deq()` methods of the `LockFreeQueue`. A node is inserted into the queue in two steps. First, a `compareAndSet()` call changes the `next` field of the node referenced by the queue's `tail` from null to the new node. Then a `compareAndSet()` call advances `tail` itself to refer to the new node. An item is removed from the queue in two steps. A `compareAndSet()` call reads the item from the node referred to by the sentinel node, and then redirects `head` from the current sentinel to the sentinel's `next` node, making the latter the new sentinel. Both `enq()` and `deq()` methods help complete unfinished `tail` updates.

We now review all the steps in detail. An enqueuer creates a new node with the new value to be enqueued (Line 10), reads `tail`, and finds the node that appears to be last (Lines 12–13). To verify that node is indeed last, it checks whether that node has a successor (Line 15). If so, the thread attempts to append the new node by calling `compareAndSet()` (Line 16). (A `compareAndSet()` is required because other threads may be trying the same thing.) If the `compareAndSet()` succeeds, the thread uses a second `compareAndSet()` to advance `tail` to the new node (Line 17). Even if this second `compareAndSet()` call fails, the thread can still return successfully because, as we will see, the call fails only if some other thread "helped" it by advancing `tail`. If the tail node has a successor (Line 20), then the method tries to "help" other threads by advancing `tail` to refer directly to the successor (Line 21) before trying again to insert its own node. This `enq()` is total, meaning that it never waits for a dequeuer. A successful `enq()` is linearized at the instant where the executing thread (or a concurrent helping thread) calls `compareAndSet()` to redirect the `tail` field to the new node at Line 21.

The `deq()` method is similar to its total counterpart from the `UnboundedQueue`. If the queue is nonempty, the dequeuer calls `compareAndSet()` to change `head` from the sentinel node to its successor, making the successor the new sentinel node. The `deq()` method makes sure that the queue is not empty in the same way as before: by checking that the `next` field of the `head` node is not *null*.

There is, however, a subtle issue in the lock-free case, depicted in Fig. 10.13: before advancing `head` one must make sure that `tail` is not left referring to the sentinel node which is about to be removed from the queue. To avoid this problem we add a test: if `head` equals `tail` (Line 32) and the (sentinel) node they refer to has a non-*null* `next` field (Line 33), then the `tail` is deemed to

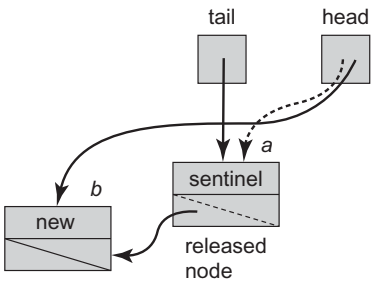

Figure 10.13 Why dequeuers must help advance `tail` in Line 36 of Fig. 10.11. Consider the scenario in which a thread enqueuing node *b* has redirected *a*'s `next` field to *b*, but has yet to redirect `tail` from *a* to *b*. If another thread starts dequeuing, it will read *b*'s value and redirect `head` from *a* to *b*, effectively removing *a* while `tail` still refers to it. To avoid this problem, the dequeuing thread must help advance `tail` from *a* to *b* before redirecting `head`.

be lagging behind. As in the `enq()` method, `deq()` then attempts to help make `tail` consistent by swinging it to the sentinel node's successor (Line 36), and only then updates `head` to remove the sentinel (Line 39). As in the partial queue, the value is read from the successor of the sentinel node (Line 38). If this method returns a value, then its linearization point occurs when it completes a successful `compareAndSet()` call at Line 39, and otherwise it is linearized at Line 33.

It is easy to check that the resulting queue is lock-free. Every method call first checks for an incomplete `enq()` call, and tries to complete it. In the worst case, all threads are trying to advance the queue's `tail` field, and one of them must succeed. A thread fails to enqueue or dequeue a node only if another thread's method call succeeds in changing the reference, so some method call always completes. As it turns out, being lock-free substantially enhances the performance of queue implementations, and the lock-free algorithms tend to outperform the most efficient blocking ones.

10.6 Memory Reclamation and the ABA Problem

Our queue implementations so far rely on the Java garbage collector to recycle nodes after they have been dequeued. What happens if we choose to do our own memory management? There are several reasons we might want to do this. Languages such as C or C++ do not provide garbage collection. Even if garbage collection is available, it is often more efficient for a class to do its own memory management, particularly if it creates and releases many small objects. Finally, if the garbage collection process is not lock-free, we might want to supply our own lock-free memory reclamation.

A natural way to recycle nodes in a lock-free manner is to have each thread maintain its own private *free list* of unused queue entries.

```
ThreadLocal<Node> freeList = new ThreadLocal<Node>() {
 protected Node initialValue() { return null; };
};
```

When an enqueuing thread needs a new node, it tries to remove one from the thread-local free list. If the free list is empty, it simply allocates a node using the **new** operator. When a dequeuing thread is ready to retire a node, it links it back onto the thread-local list. Because the list is thread-local, there is no need for expensive synchronization. This design works well, as long as each thread

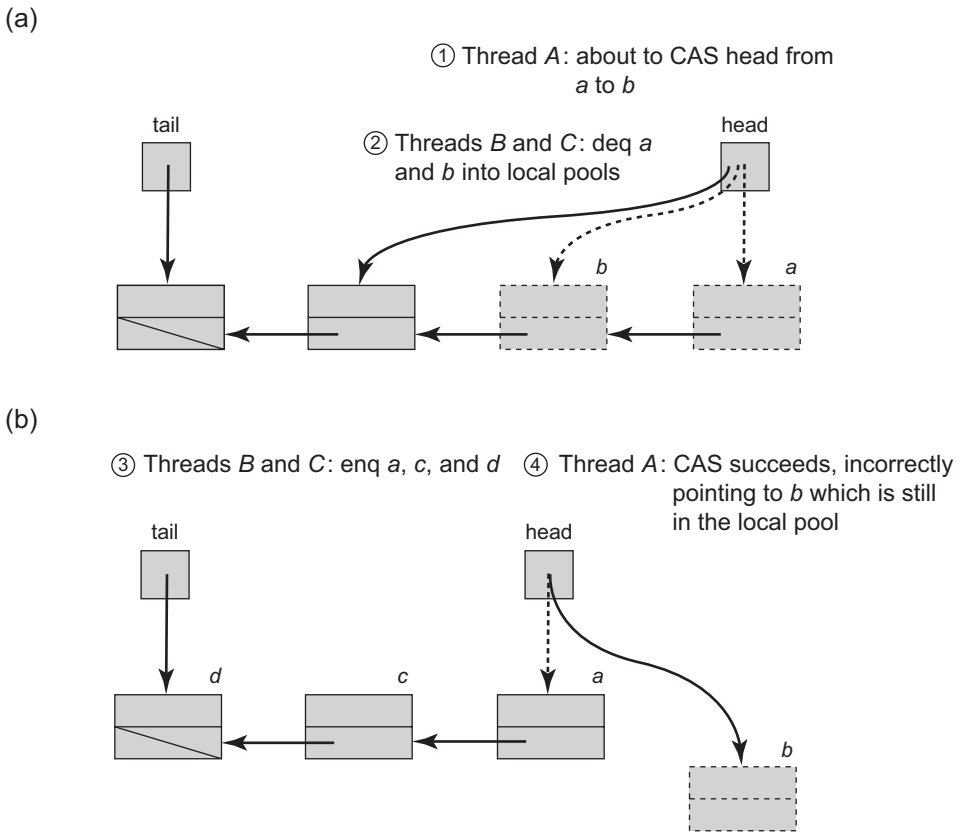

Figure 10.14 An ABA scenario: Assume that we use local pools of recycled nodes in our lock-free queue algorithm. In Part (a), the dequeuer thread *A* of Fig. 10.11 observes that the sentinel node is *a*, and next node is *b*. (Step 1) It then prepares to update `head` by applying a `compareAndSet()` with old value *a* and new value *b*. (Step 2) Suppose however, that before it takes another step, other threads dequeue *b*, then its successor, placing both *a* and *b* in the free pool. In Part (b) (Step 3) node *a* is reused, and eventually reappears as the sentinel node in the queue. (Step 4) thread *A* now wakes up, calls `compareAndSet()`, and succeeds in setting `head` to *b*, since the old value of `head` is indeed *a*. Now, `head` is incorrectly set to a recycled node.

performs roughly the same number of enqueues and dequeues. If there is an imbalance, then there may be a need for more complex techniques, such as periodically stealing nodes from other threads.

Surprisingly, perhaps, the lock-free queue will not work if nodes are recycled in the most straightforward way. Consider the scenario depicted in Fig. 10.14. In Part (a) of Fig. 10.14, the dequeuing thread A observes the sentinel node is *a*, and the next node is *b*. It then prepares to update `head` by applying a `compareAndSet()` with old value *a* and new value *b*. Before it takes another step, other threads dequeue *b* and its successor, placing both *a* and *b* in the free pool. Node *a* is recycled, and eventually reappears as the sentinel node in the queue, as depicted in Part (b) of Fig. 10.14. The thread now wakes up, calls `compareAndSet()`, and succeeds, since the old value of the `head` is indeed *a*. Unfortunately, it has redirected `head` to a recycled node!

This phenomenon is called the "ABA" problem. It shows up often, especially in dynamic memory algorithms that use conditional synchronization operations such as `compareAndSet()`. Typically, a reference about to be modified by a `compareAndSet()` changes from *a*, to *b*, and back to *a* again. As a result, the `compareAndSet()` call succeeds even though its effect on the data structure has changed, and no longer has the desired effect.

One straightforward way to fix this problem is to tag each atomic reference with a unique *stamp*. As described in detail in Pragma 10.6.1, an `AtomicStampedReference<T>` object encapsulates both a reference to an object of Type `T` and an integer `stamp`. These fields can be atomically updated either together or individually.

Pragma 10.6.1. The `AtomicStampedReference<T>` class encapsulates both a reference to an object of Type `T` and an integer *stamp*. It generalizes the `AtomicMarkableReference<T>` class (Pragma 9.8.1), replacing the Boolean *mark* with an integer stamp.

We usually use this stamp to avoid the ABA problem, incrementing the value of the stamp each time we modify the object, although sometimes, as in the `LockFreeExchanger` class of Chapter 11, we use the stamp to hold one of a finite set of states.

The stamp and reference fields can be updated atomically, either together or individually. For example, the `compareAndSet()` method tests expected reference and stamp values, and if both tests succeed, replaces them with updated reference and stamp values. As shorthand, the `attemptStamp()` method tests an expected reference value and if the test succeeds, replaces it with a new stamp value. The `get()` method has an unusual interface: it returns the object's reference value and stores the stamp value in an integer array argument. Fig. 10.15 illustrates the signatures for these methods.

```
1  public boolean compareAndSet(T expectedReference,
2                               T newReference,
3                               int expectedStamp,
4                               int newStamp);
5  public T get(int[] stampHolder);
6  public void set(T newReference, int newStamp);
```

Figure 10.15 The `AtomicStampedReference<T>` class: the `compareAndSet()` and `get()` methods. The `compareAndSet()` method tests and updates both the stamp and reference fields, the `get()` method returns the encapsulated reference and stores the stamp at position 0 in the argument array, and `set()` updates the encapsulated reference and the stamp.

In a language like C or C++, one could provide this functionality efficiently in a 64-bit architecture by "stealing" bits from pointers, although a 32-bit architecture would probably require a level of indirection.

As shown in Fig. 10.16, each time through the loop, `deq()` reads both the reference and stamp values for the first, next, and last nodes (Lines 6–8). It uses `compareAndSet()` to compare both the reference and the stamp (Line 17).

```
1   public T deq() throws EmptyException {
2     int[] lastStamp = new int[1];
3     int[] firstStamp = new int[1];
4     int[] nextStamp = new int[1];
5     while (true) {
6       Node first = head.get(firstStamp);
7       Node last = tail.get(lastStamp);
8       Node next = first.next.get(nextStamp);
9         if (first == last) {
10          if (next == null) {
11            throw new EmptyException();
12          }
13          tail.compareAndSet(last, next,
14              lastStamp[0], lastStamp[0]+1);
15        } else {
16          T value = next.value;
17          if (head.compareAndSet(first, next, firstStamp[0],
                firstStamp[0]+1)) {
18            free(first);
19            return value;
20          }
21        }
22    }
23  }
```

Figure 10.16 The `LockFreeQueueRecycle<T>` class: the `deq()` method uses stamps to avoid ABA.

It increments the stamp each time it uses `compareAndSet()` to update a reference (Lines 14 and 17).[1]

The ABA problem can occur in many synchronization scenarios, not just those involving conditional synchronization. For example, it can occur when using only loads and stores. Conditional synchronization operations such as *load-linked/store-conditional*, available on some architectures (see Appendix B), avoid ABA by testing not whether a value is the same at two points in time, but whether the value has ever changed between those points.

10.6.1 A Naïve Synchronous Queue

We now turn our attention to an even tighter kind of synchronization. One or more *producer* threads produce items to be removed, in first-in-first-out order, by one or more *consumer* threads. Here, however, producers and consumers *rendezvous* with one another: a producer that puts an item in the queue blocks until that item is removed by a consumer, and vice versa. Such rendezvous synchronization is built into languages such as CSP and Ada.

Fig. 10.17 illustrates the `SynchronousQueue<T>` class, a straightforward monitor-based synchronous queue implementation. It has the following fields: `item` is the first item waiting to be dequeued, `enqueuing` is a Boolean value used by enqueuers to synchronize among themselves, `lock` is the lock used for mutual exclusion, and `condition` is used to block partial methods. If the `enq()` method finds `enqueuing` to be *true* (Line 10) then another enqueuer has supplied an item and is waiting to rendezvous with a dequeuer, so the enqueuer repeatedly releases the lock, sleeps, and checks whether `enqueuing` has become *false* (Line 11). When this condition is satisfied, the enqueuer sets `enqueuing` to *true*, which locks out other enqueuers until the current rendezvous is complete, and sets `item` to refer to the new item (Lines 12–13). It then notifies any waiting threads (Line 14), and waits until `item` becomes *null* (Lines 15–16). When the wait is over, the rendezvous has occurred, so the enqueuer sets `enqueuing` to *false*, notifies any waiting threads, and returns (Lines 17 and 19).

The `deq()` method simply waits until `item` is non-*null* (Lines 26–27), records the item, sets the `item` field to *null*, and notifies any waiting threads before returning the item (Lines 28–31).

While the design of the queue is relatively simple, it incurs a high synchronization cost. At every point where one thread might wake up another, both enqueuers and dequeuers wake up all waiting threads, leading to a number of wakeups quadratic in the number of waiting threads. While it is possible to use condition objects to reduce the number of wakeups, it is still necessary to block on every call, which is expensive.

---

1 We ignore the remote possibility that the stamp could wrap around and cause an error.

```
1  public class SynchronousQueue<T> {
2    T item = null;
3    boolean enqueuing;
4    Lock lock;
5    Condition condition;
6    ...
7    public void enq(T value) {
8      lock.lock();
9      try {
10       while (enqueuing)
11         condition.await();
12       enqueuing = true;
13       item = value;
14       condition.signalAll();
15       while (item != null)
16         condition.await();
17       enqueuing = false;
18       condition.signalAll();
19     } finally {
20       lock.unlock();
21     }
22   }
23   public T deq() {
24     lock.lock();
25     try {
26       while (item == null)
27         condition.await();
28       T t = item;
29       item = null;
30       condition.signalAll();
31       return t;
32     } finally {
33       lock.unlock();
34     }
35   }
36 }
```

Figure 10.17 The SynchronousQueue<T> class.

10.7 Dual Data Structures

To reduce the synchronization overheads of the synchronous queue, we consider an alternative synchronous queue implementation that splits `enq()` and `deq()` methods into two steps. Here is how a dequeuer tries to remove an item from an empty queue. In the first step, it puts a *reservation* object in the queue, indicating that the dequeuer is waiting for an enqueuer with which to rendezvous. The dequeuer then spins on a flag in the reservation. Later, when an enqueuer discovers the reservation, it *fulfills* the reservation by depositing an item and notifying

the dequeuer by setting the reservation's flag. Similarly, an enqueuer can wait for a rendezvous partner by creating its own reservation, and spinning on the reservation's flag. At any time the queue itself contains either `enq()` reservations, `deq()` reservations, or it is empty.

This structure is called a *dual data structure*, since the methods take effect in two stages, reservation and fulfillment. It has a number of nice properties. First, waiting threads can spin on a locally cached flag, which we have seen is essential for scalability. Second, it ensures fairness in a natural way. Reservations are queued in the order they arrive, ensuring that requests are fulfilled in the same order. Note that this data structure is linearizable, since each partial method call can be ordered when it is fulfilled.

The queue is implemented as a list of nodes, where a node represents either an item waiting to be dequeued, or a reservation waiting to be fulfilled (Fig. 10.18). A node's `type` field indicates which. At any time, all queue nodes have the same type: either the queue consists entirely of items waiting to be dequeued, or entirely of reservations waiting to be fulfilled.

When an item is enqueued, the node's `item` field holds the item, which is reset to *null* when that item is dequeued. When a reservation is enqueued, the node's `item` field is *null*, and is reset to an item when fulfilled by an enqueuer.

Fig. 10.19 shows the `SynchronousDualQueue`'s constructor and `enq()` method. (The `deq()` method is symmetric.) Just like the earlier queues we have considered, the `head` field always refers to a *sentinel* node that serves as a place-holder, and whose actual value is unimportant. The queue is empty when `head` and `tail` agree. The constructor creates a sentinel node with an arbitrary value, referred to by both `head` and `tail`.

The `enq()` method first checks whether the queue is empty or whether it contains enqueued items waiting to be dequeued (Line 10). If so, then just as in the lock-free queue, the method reads the queue's `tail` field (Line 11), and checks that the values read are consistent (Line 12). If the `tail` field does not refer to the last node in the queue, then the method advances the `tail` field and starts over (Lines 13–14). Otherwise, the `enq()` method tries to append the new node to the end of the queue by resetting the tail node's `next` field to refer to the new

```
1   private enum NodeType {ITEM, RESERVATION};
2   private class Node {
3     volatile NodeType type;
4     volatile AtomicReference<T> item;
5     volatile AtomicReference<Node> next;
6     Node(T myItem, NodeType myType) {
7       item = new AtomicReference<T>(myItem);
8       next = new AtomicReference<Node>(null);
9       type = myType;
10    }
11  }
```

Figure 10.18 The `SynchronousDualQueue<T>` class: queue node.

```
1   public SynchronousDualQueue() {
2     Node sentinel = new Node(null, NodeType.ITEM);
3     head = new AtomicReference<Node>(sentinel);
4     tail = new AtomicReference<Node>(sentinel);
5   }
6   public void enq(T e) {
7     Node offer = new Node(e, NodeType.ITEM);
8     while (true) {
9       Node t = tail.get(), h = head.get();
10      if (h == t || t.type == NodeType.ITEM) {
11        Node n = t.next.get();
12        if (t == tail.get()) {
13          if (n != null) {
14            tail.compareAndSet(t, n);
15          } else if (t.next.compareAndSet(n, offer)) {
16            tail.compareAndSet(t, offer);
17            while (offer.item.get() == e);
18            h = head.get();
19            if (offer == h.next.get())
20              head.compareAndSet(h, offer);
21            return;
22          }
23        }
24      } else {
25        Node n = h.next.get();
26        if (t != tail.get() || h != head.get() || n == null) {
27          continue;
28        }
29        boolean success = n.item.compareAndSet(null, e);
30        head.compareAndSet(h, n);
31        if (success)
32          return;
33      }
34    }
35  }
```

Figure 10.19 The `SynchronousDualQueue<T>` class: `enq()` method and constructor.

node (Line 15). If it succeeds, it tries to advance the tail to the newly appended node (Line 16), and then spins, waiting for a dequeuer to announce that it has dequeued the item by setting the node's `item` field to *null*. Once the item is dequeued, the method tries to clean up by making its node the new sentinel. This last step serves only to enhance performance, because the implementation remains correct, whether or not the method advances the `head` reference.

If, however, the `enq()` method discovers that the queue contains dequeuers' reservations waiting to be fulfilled, then it tries to find a reservation to fulfill. Since the queue's `head` node is a sentinel with no meaningful value, `enq()` reads the head's successor (Line 25), checks that the values it has read are consistent (Lines 26–28), and tries to switch that node's `item` field from *null* to the item being enqueued. Whether or not this step succeeds, the method tries to advance

head (Line 30). If the compareAndSet() call succeeds (Line 29), the method returns; otherwise it retries.

10.8 Chapter Notes

The partial queue employs a mixture of techniques adapted from Doug Lea [98] and from an algorithm by Maged Michael and Michael Scott [115]. The lock-free queue is a slightly simplified version of a queue algorithm by Maged Michael and Michael Scott [115]. The synchronous queue implementations are adapted from algorithms by Bill Scherer, Doug Lea, and Michael Scott [136].

10.9 Exercises

Exercise 119. Change the SynchronousDualQueue<T> class to work correctly with *null* items.

Exercise 120. Consider the simple lock-free queue for a single enqueuer and a single dequeuer, described earlier in Chapter 3. The queue is presented in Fig. 10.20.

```
1  class TwoThreadLockFreeQueue<T> {
2    int head = 0, tail = 0;
3    T[] items;
4    public TwoThreadLockFreeQueue(int capacity) {
5      head = 0; tail = 0;
6      items = (T[]) new Object[capacity];
7    }
8    public void enq(T x) {
9      while (tail - head == items.length) {};
10     items[tail % items.length] = x;
11     tail++;
12   }
13   public Object deq() {
14     while (tail - head == 0) {};
15     Object x = items[head % items.length];
16     head++;
17     return x;
18   }
19 }
```

Figure 10.20 A Lock-free FIFO queue with blocking semantics for a single enqueuer and single dequeuer. The queue is implemented in an array. Initially the head and tail fields are equal and the queue is empty. If the head and tail differ by capacity, then the queue is full. The enq() method reads the head field, and if the queue is full, it repeatedly checks the head until the queue is no longer full. It then stores the object in the array, and increments the tail field. The deq() method works in a symmetric way.

This queue is blocking, that is, removing an item from an empty queue or inserting an item to a full one causes the threads to block (spin). The surprising thing about this queue is that it requires only loads and stores and not a more powerful read–modify–write synchronization operation. Does it however require the use of a memory barrier? If not, explain, and if so, where in the code is such a barrier needed and why?

Exercise 121. Design a bounded lock-based queue implementation using an array instead of a linked list.

1. Allow parallelism by using two separate locks for `head` and `tail`.
2. Try to transform your algorithm to be lock-free. Where do you run into difficulty?

Exercise 122. Consider the unbounded lock-based queue's `deq`() method in Fig. 10.8. Is it necessary to hold the lock when checking that the queue is not empty? Explain.

Exercise 123. In Dante's *Inferno*, he describes a visit to Hell. In a very recently discovered chapter, he encounters five people sitting at a table with a pot of stew in the middle. Although each one holds a spoon that reaches the pot, each spoon's handle is much longer than each person's arm, so no one can feed him- or herself. They are famished and desperate.

Dante then suggests "why do not you feed one another?"

The rest of the chapter is lost.

1. Write an algorithm to allow these unfortunates to feed one another. Two or more people may not feed the same person at the same time. Your algorithm must be, well, starvation-free.
2. Discuss the advantages and disadvantages of your algorithm. Is it centralized, decentralized, high or low in contention, deterministic or randomized?

Exercise 124. Consider the linearization points of the `enq`() and `deq`() methods of the lock-free queue:

1. Can we choose the point at which the returned value is read from a node as the linearization point of a successful `deq`()?
2. Can we choose the linearization point of the `enq`() method to be the point at which the `tail` field is updated, possibly by other threads (consider if it is within the `enq`()'s execution interval)? Argue your case.

Exercise 125. Consider the unbounded queue implementation shown in Fig. 10.21. This queue is blocking, meaning that the `deq`() method does not return until it has found an item to dequeue.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    ...
5    public void enq(T x) {
6      int i = tail.getAndIncrement();
7      items[i].set(x);
8    }
9    public T deq() {
10     while (true) {
11      int range = tail.get();
12      for (int i = 0; i < range; i++) {
13        T value = items[i].getAndSet(null);
14        if (value != null) {
15          return value;
16        }
17       }
18     }
19   }
20 }
```

Figure 10.21 Queue used in Exercise 125.

The queue has two fields: `items` is a very large array, and `tail` is the index of the next unused element in the array.

1. Are the `enq()` and `deq()` methods wait-free? If not, are they lock-free? Explain.
2. Identify the linearization points for `enq()` and `deq()`. (Careful! They may be execution-dependent.)

This page intentionally left blank

11 Concurrent Stacks and Elimination

11.1 Introduction

The `Stack<T>` class is a collection of items (of type `T`) that provides `push`() and `pop`() methods satisfying the *last-in-first-out* (LIFO) property: the last item pushed is the first popped. This chapter considers how to implement concurrent stacks. At first glance, stacks seem to provide little opportunity for concurrency, because `push`() and `pop`() calls seem to need to synchronize at the top of the stack.

Surprisingly, perhaps, stacks are not inherently sequential. In this chapter, we show how to implement concurrent stacks that can achieve a high degree of parallelism. As a first step, we consider how to build a lock-free stack in which pushes and pops synchronize at a single location.

11.2 An Unbounded Lock-Free Stack

Fig. 11.1 shows a concurrent `LockFreeStack` class, whose code appears in Figs. 11.2, 11.3 and 11.4. The lock-free stack is a linked list, where the `top` field points to the first node (or *null* if the stack is empty.) For simplicity, we usually assume it is illegal to add a *null* value to a stack.

A `pop`() call that tries to remove an item from an empty stack throws an exception. A `push`() method creates a new node (Line 13), and then calls `tryPush`() to try to swing the `top` reference from the current top-of-stack to its successor. If `tryPush`() succeeds, `push`() returns, and if not, the `tryPush`() attempt is repeated after backing off. The `pop`() method calls `tryPop`(), which uses `compareAndSet()` to try to remove the first node from the stack. If it succeeds, it returns the node, otherwise it returns *null*. (It throws an exception if the stack

The Art of Multiprocessor Programming. DOI: 10.1016/B978-0-12-397337-5.00011-3

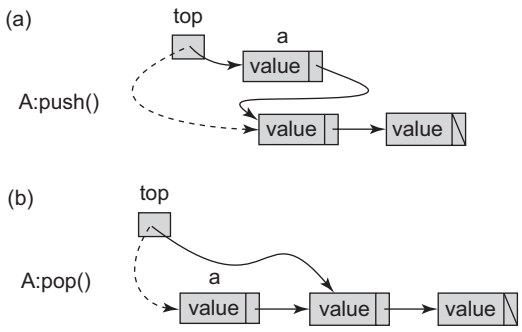

Figure 11.1 A Lock-free stack. In Part (a) a thread pushes value *a* into the stack by applying a `compareAndSet()` to the `top` field. In Part (b) a thread pops value *a* from the stack by applying a `compareAndSet()` to the `top` field.

```
1  public class LockFreeStack<T> {
2    AtomicReference<Node> top = new AtomicReference<Node>(null);
3    static final int MIN_DELAY = ...;
4    static final int MAX_DELAY = ...;
5    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
6
7    protected boolean tryPush(Node node){
8      Node oldTop = top.get();
9      node.next = oldTop;
10     return(top.compareAndSet(oldTop, node));
11   }
12   public void push(T value) {
13     Node node = new Node(value);
14     while (true) {
15       if (tryPush(node)) {
16         return;
17       } else {
18         backoff.backoff();
19       }
20     }
21   }
```

Figure 11.2 The `LockFreeStack<T>` class: in the `push()` method, threads alternate between trying to alter the `top` reference by calling `tryPush()`, and backing off using the `Backoff` class from Fig. 7.5 of Chapter 7.

is empty.) The `tryPop()` method is called until it succeeds, at which point `pop()` returns the value from the removed node.

As we have seen in Chapter 7, one can significantly reduce contention at the `top` field using exponential backoff (see Fig. 7.5 of Chapter 7). Accordingly, both

```
1  public class Node {
2      public T value;
3      public Node next;
4      public Node(T value) {
5        value = value;
6        next = null;
7      }
8  }
```

Figure 11.3 Lock-free stack list node.

```
1    protected Node tryPop() throws EmptyException {
2      Node oldTop = top.get();
3      if (oldTop == null) {
4        throw new EmptyException();
5      }
6      Node newTop = oldTop.next;
7      if (top.compareAndSet(oldTop, newTop)) {
8        return oldTop;
9      } else {
10       return null;
11     }
12   }
13   public T pop() throws EmptyException {
14     while (true) {
15       Node returnNode = tryPop();
16       if (returnNode != null) {
17         return returnNode.value;
18       } else {
19         backoff.backoff();
20       }
21     }
22   }
```

Figure 11.4 The LockFreeStack<T> class: The pop() method alternates between trying to change the top field and backing off.

the push() and pop() methods back off after an unsuccessful call to tryPush() or tryPop().

This implementation is lock-free because a thread fails to complete a push() or pop() method call only if there were infinitely many successful calls that modified the top of the stack. The linearization point of both the push() and the pop() methods is the successful compareAndSet(), or the throwing of the exception, in Line 3, in case of a pop() on an empty stack. Note that the compareAndSet() call by pop() does not have an ABA problem (see Chapter 10) because the Java garbage collector ensures that a node cannot be reused by one thread, as long as that node is accessible to another thread. Designing a lock-free stack that avoids the ABA problem without a garbage collector is left as an exercise.

11.3 Elimination

The `LockFreeStack` implementation scales poorly, not so much because the stack's `top` field is a source of *contention*, but primarily because it is a *sequential bottleneck*: method calls can proceed only one after the other, ordered by successful `compareAndSet()` calls applied to the stack's `top` field.

Although exponential backoff can significantly reduce contention, it does nothing to alleviate the sequential bottleneck. To make the stack parallel, we exploit this simple observation: if a `push()` is immediately followed by a `pop()`, the two operations cancel out, and the stack's state does not change. It is as if both operations never happened. If one could somehow cause concurrent pairs of pushes and pops to cancel, then threads calling `push()` could exchange values with threads calling `pop()`, without ever modifying the stack itself. These two calls would *eliminate* one another.

As depicted in Fig. 11.5, threads eliminate one another through an `EliminationArray` in which threads pick random array entries to try to meet complementary calls. Pairs of complementary `push()` and `pop()` calls exchange values and return. A thread whose call cannot be eliminated, either because it has failed to find a partner, or found a partner with the wrong kind of method call (such as a `push()` meeting a `push()`), can either try again to eliminate at a new location, or can access the shared `LockFreeStack`. The combined data structure, array, and shared stack, is linearizable because the shared stack is linearizable, and the eliminated calls can be ordered as if they happened at the point in which they exchanged values.

We can use the `EliminationArray` as a backoff scheme on a shared `LockFreeStack`. Each thread first accesses the `LockFreeStack`, and if it fails

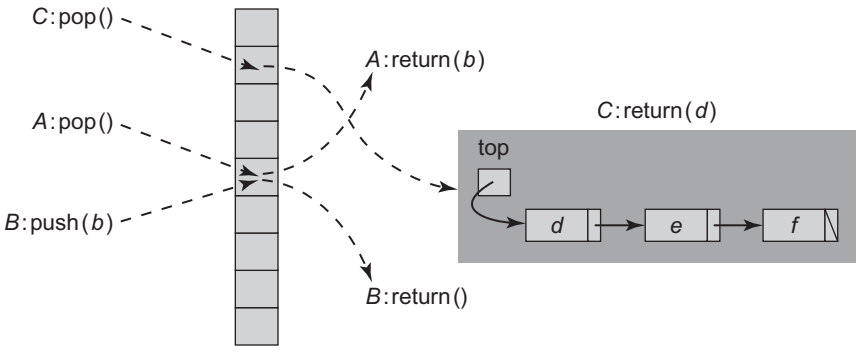

Figure 11.5 The `EliminationBackoffStack<T>` class. Each thread selects a random location in the array. If thread *A*'s `pop()` and *B*'s `push()` calls arrive at the same location at about the same time, then they exchange values without accessing the shared `LockFreeStack`. Thread *C* that does not meet another thread eventually pops the shared `LockFreeStack`.

to complete its call (that is, the `compareAndSet()` attempt fails), it attempts to eliminate its call using the array instead of simply backing off. If it fails to eliminate itself, it calls the `LockFreeStack` again, and so on. We call this structure an `EliminationBackoffStack`.

11.4 The Elimination Backoff Stack

Here is how to construct an `EliminationBackoffStack`, a lock-free linearizable stack implementation.

We are reminded of a story about two friends who are discussing politics on election day, each trying, to no avail, to convince the other to switch sides.

Finally, one says to the other: "Look, it's clear that we are unalterably opposed on every political issue. Our votes will surely cancel out. Why not save ourselves some time and both agree to not vote today?"

The other agrees enthusiastically and they part.

Shortly after that, a friend of the first one who had heard the conversation says, "That was a sporting offer you made."

"Not really," says the second. "This is the third time I've done this today."

The principle behind our construction is the same. We wish to allow threads with pushes and pops to coordinate and cancel out, but must avoid a situation in which a thread can make a sporting offer to more than one other thread. We do so by implementing the `EliminationArray` using coordination structures called *exchangers*, objects that allow exactly two threads (and no more) to rendezvous and exchange values.

We already saw how to exchange values using locks in the synchronous queue of Chapter 10. Here, we need a lock-free exchange, one in which threads spin rather than block, as we expect them to wait only for very short durations.

11.4.1 A Lock-Free Exchanger

A `LockFreeExchanger<T>` object permits two threads to exchange values of type `T`. If thread *A* calls the object's `exchange()` method with argument *a*, and *B* calls the same object's `exchange()` method with argument *b*, then *A*'s call will return value *b* and vice versa. On a high level, the exchanger works by having the first thread arrive to write its value, and spin until a second arrives. The second then detects that the first is waiting, reads its value, and signals the exchange. They each have now read the other's value, and can return. The first thread's call may timeout if the second does not show up, allowing it to proceed and leave the exchanger, if it is unable to exchange a value within a reasonable duration.

```
1  public class LockFreeExchanger<T> {
2    static final int EMPTY = ..., WAITING = ..., BUSY = ...;
3    AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
4    public T exchange(T myItem, long timeout, TimeUnit unit)
5      throws TimeoutException {
6      long nanos = unit.toNanos(timeout);
7      long timeBound = System.nanoTime() + nanos;
8      int[] stampHolder = {EMPTY};
9      while (true) {
10       if (System.nanoTime() > timeBound)
11         throw new TimeoutException();
12       T yrItem = slot.get(stampHolder);
13       int stamp = stampHolder[0];
14       switch(stamp) {
15       case EMPTY:
16         if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17           while (System.nanoTime() < timeBound){
18             yrItem = slot.get(stampHolder);
19             if (stampHolder[0] == BUSY) {
20               slot.set(null, EMPTY);
21               return yrItem;
22             }
23           }
24           if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25             throw new TimeoutException();
26           } else {
27             yrItem = slot.get(stampHolder);
28             slot.set(null, EMPTY);
29             return yrItem;
30           }
31         }
32         break;
33       case WAITING:
34         if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
35           return yrItem;
36         break;
37       case BUSY:
38         break;
39       default: // impossible
40         ...
41       }
42     }
43   }
44 }
```

Figure 11.6 The LockFreeExchanger<T> Class.

The LockFreeExchanger<T> class appears in Fig. 11.6. It has a single AtomicStampedReference<T> field,[1] slot. The exchanger has three possible states: EMPTY, BUSY, or WAITING. The reference's stamp records the exchanger's state (Line 14). The exchanger's main loop continues until the timeout limit

1 See Chapter 10, Pragma 10.6.1.

passes, when it throws an exception (Line 10). In the meantime, a thread reads the state of the `slot` (Line 12) and proceeds as follows:

- If the state is `EMPTY`, then the thread tries to place its item in the slot and set the state to `WAITING` using a `compareAndSet()` (Line 16). If it fails, then some other thread succeeds and it retries. If it was successful (Line 17), then its item is in the slot and the state is `WAITING`, so it spins, waiting for another thread to complete the exchange. If another thread shows up, it will take the item in the slot, replace it with its own, and set the state to `BUSY` (Line 19), indicating to the waiting thread that the exchange is complete. The waiting thread will consume the item and reset the state to `EMPTY`. Resetting to `EMPTY` can be done using a simple write because the waiting thread is the only one that can change the state from `BUSY` to `EMPTY` (Line 20). If no other thread shows up, the waiting thread needs to reset the state of the slot to `EMPTY`. This change requires a `compareAndSet()` because other threads might be attempting to exchange by setting the state from `WAITING` to `BUSY` (Line 24). If the call is successful, it raises a timeout exception. If, however, the call fails, some exchanging thread must have shown up, so the waiting thread completes the exchange (Line 26).
- If the state is `WAITING`, then some thread is waiting and the slot contains its item. The thread takes the item, and tries to replace it with its own by changing the state from `WAITING` to `BUSY` using a `compareAndSet()` (Line 34). It may fail if another thread succeeds, or the other thread resets the state to `EMPTY` following a timeout. If so, the thread must retry. If it does succeed changing the state to `BUSY`, then it can return the item.
- If the state is `BUSY` then two other threads are currently using the slot for an exchange and the thread must retry (Line 37).

Notice that the algorithm allows the inserted item to be *null*, something used later in the elimination array construction. There is no ABA problem because the `compareAndSet()` call that changes the state never inspects the item. A successful exchange's linearization point occurs when the second thread to arrive changes the state from `WAITING` to `BUSY` (Line 34). At this point both `exchange()` calls overlap, and the exchange is committed to being successful. An unsuccessful exchange's linearization point occurs when the timeout exception is thrown.

The algorithm is lock-free because overlapping `exchange()` calls with sufficient time to exchange will fail only if other exchanges are repeatedly succeeding. Clearly, too short an exchange time can cause a thread never to succeed, so care must be taken when choosing timeout durations.

11.4.2 The Elimination Array

An `EliminationArray` is implemented as an array of `Exchanger` objects of maximal size `capacity`. A thread attempting to perform an exchange picks an array entry at random, and calls that entry's `exchange()` method, providing

```
1  public class EliminationArray<T> {
2    private static final int duration = ...;
3    LockFreeExchanger<T>[] exchanger;
4    Random random;
5    public EliminationArray(int capacity) {
6      exchanger = (LockFreeExchanger<T>[]) new LockFreeExchanger[capacity];
7      for (int i = 0; i < capacity; i++) {
8        exchanger[i] = new LockFreeExchanger<T>();
9      }
10     random = new Random();
11   }
12   public T visit(T value, int range) throws TimeoutException {
13     int slot = random.nextInt(range);
14     return (exchanger[slot].exchange(value, duration,
15             TimeUnit.MILLISECONDS));
16   }
17 }
```

Figure 11.7 The `EliminationArray<T>` class: in each visit, a thread can choose dynamically the sub-range of the array from which it will will randomly select a slot.

its own input as an exchange value with another thread. The code for the `EliminationArray` appears in Fig. 11.7. The constructor takes as an argument the `capacity` of the array (the number of distinct exchangers). The `EliminationArray` class provides a single method, `visit()`, which takes timeout arguments. (Following the conventions used in the java.util.concurrent package, a timeout is expressed as a number and a time unit.) The `visit()` call takes a value of type `T` and either returns the value input by its exchange partner, or throws an exception if the timeout expires without exchanging a value with another thread. At any point in time, each thread will select a random location in a subrange of the array (Line 13). This subrange will be determined dynamically based on the load on the data structure, and will be passed as a parameter to the `visit()` method.

The `EliminationBackoffStack` is a subclass of `LockFreeStack` that overrides the `push()` and `pop()` methods, and adds an `EliminationArray` field. Figs. 11.8 and 11.9 show the new `push()` and `pop()` methods. Upon failure of a `tryPush()` or `tryPop()` attempt, instead of simply backing off, these methods try to use the `EliminationArray` to exchange values (Lines 15 and 34). A `push()` call calls `visit()` with its input value as argument, and a `pop()` call with *null* as argument. Both `push()` and `pop()` have a thread-local `RangePolicy` object that determines the `EliminationArray` subrange to be used.

When `push()` calls `visit()`, it selects a random array entry within its range and attempts to exchange a value with another thread. If the exchange is successful, the pushing thread checks whether the value was exchanged with a `pop()` method (Line 18) by testing if the value exchanged was *null*. (Recall that `pop()` always offers *null* to the exchanger while `push()` always offers a non-*null* value.) Symmetrically, when `pop()` calls `visit()`, it attempts an exchange, and if the

```
1  public class EliminationBackoffStack<T> extends LockFreeStack<T> {
2    static final int capacity = ...;
3    EliminationArray<T> eliminationArray = new EliminationArray<T>(capacity);
4    static ThreadLocal<RangePolicy> policy = new ThreadLocal<RangePolicy>() {
5      protected synchronized RangePolicy initialValue() {
6        return new RangePolicy();
7      }
8
9    public void push(T value) {
10     RangePolicy rangePolicy = policy.get();
11     Node node = new Node(value);
12     while (true) {
13       if (tryPush(node)) {
14         return;
15       } else try {
16         T otherValue = eliminationArray.visit
17                           (value, rangePolicy.getRange());
18         if (otherValue == null) {
19           rangePolicy.recordEliminationSuccess();
20           return; // exchanged with pop
21         }
22       } catch (TimeoutException ex) {
23         rangePolicy.recordEliminationTimeout();
24       }
25     }
26   }
27 }
```

Figure 11.8 The EliminationBackoffStack<T> class: this push() method overrides the LockFreeStack push() method. Instead of using a simple Backoff class, it uses an EliminationArray and a dynamic RangePolicy to select the subrange of the array within which to eliminate.

```
28   public T pop() throws EmptyException {
29     RangePolicy rangePolicy = policy.get();
30     while (true) {
31       Node returnNode = tryPop();
32       if (returnNode != null) {
33         return returnNode.value;
34       } else try {
35         T otherValue = eliminationArray.visit(null, rangePolicy.getRange());
36         if (otherValue != null) {
37           rangePolicy.recordEliminationSuccess();
38           return otherValue;
39         }
40       } catch (TimeoutException ex) {
41         rangePolicy.recordEliminationTimeout();
42       }
43     }
44   }
```

Figure 11.9 The EliminationBackoffStack<T> class: this pop() method overrides the LockFreeStack push() method.

exchange is successful it checks (Line 36) whether the value was exchanged with a `push()` call by checking whether it is not *null*.

It is possible that the exchange will be unsuccessful, either because no exchange took place (the call to `visit()` timed out) or because the exchange was with the same type of operation (such as a `pop()` with a `pop()`). For brevity, we choose a simple approach to deal with such cases: we retry the `tryPush()` or `tryPop()` calls (Lines 13 and 31).

One important parameter is the range of the `EliminationArray` from which a thread selects an `Exchanger` location. A smaller range will allow a greater chance of a successful collision when there are few threads, while a larger range will lower the chances of threads waiting on a busy `Exchanger` (recall that an `Exchanger` can only handle one exchange at a time). Thus, if few threads access the array, they should choose smaller ranges, and as the number of threads increase, so should the range. One can control the range dynamically using a `RangePolicy` object that records both successful exchanges (as in Line 37) and timeout failures (Line 40). We ignore exchanges that fail because the operations do not match (such as `push()` with `push()`), because they account for a fixed fraction of the exchanges for any given distribution of `push()` and `pop()` calls. One simple policy is to shrink the range as the number of failures increases and vice versa.

There are many other possible policies. For example, one can devise a more elaborate range selection policy, vary the delays on the exchangers dynamically, add additional backoff delays before accessing the shared stack, and control whether to access the shared stack or the array dynamically. We leave these as exercises.

The `EliminationBackoffStack` is a linearizable stack: any successful `push()` or `pop()` call that completes by accessing the `LockFreeStack` can be linearized at the point of its `LockFreeStack` access. Any pair of eliminated `push()` and `pop()` calls can be linearized when they collide. As noted earlier, the method calls completed through elimination do not affect the linearizability of those completed in the `LockFreeStack`, because they could have taken effect in any state of the `LockFreeStack`, and having taken effect, the state of the `LockFreeStack` would not have changed.

Because the `EliminationArray` is effectively used as a backoff scheme, we expect it to deliver performance comparable to the `LockFreeStack` at low loads. Unlike the `LockFreeStack`, it has the potential to scale. As the load increases, the number of successful eliminations will grow, allowing many operations to complete in parallel. Moreover, contention at the `LockFreeStack` is reduced because eliminated operations never access the stack.

11.5 Chapter Notes

The `LockFreeStack` is credited to Treiber [145]. Actually it predates Treiber's report in 1986. It was probably invented in the early 1970s to motivate the

CAS operation on the IBM 370. The `EliminationBackoffStack` is due to Danny Hendler, Nir Shavit, and Lena Yerushalmi [57]. An efficient exchanger, which quite interestingly uses an elimination array, was introduced by Doug Lea, Michael Scott, and Bill Scherer [136]. A variant of this exchanger appears in the Java Concurrency Package. The `EliminationBackoffStack` we present here is modular, making use of exchangers, but somewhat inefficient. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit present a highly effective implementation of an `EliminationArray` [120].

11.6 Exercises

Exercise 126. Design an unbounded lock-based `Stack<T>` implementation based on a linked list.

Exercise 127. Design a bounded lock-based `Stack<T>` using an array.

1. Use a single lock and a bounded array.
2. Try to make your algorithm lock-free. Where do you run into difficulty?

Exercise 128. Modify the unbounded lock-free stack of Section 11.2 to work in the absence of a garbage collector. Create a thread-local pool of preallocated nodes and recycle them. To avoid the ABA problem, consider using the `AtomicStampedReference<T>` class from `java.util.concurrent.atomic` that encapsulates both a reference and an integer *stamp*.

Exercise 129. Discuss the backoff policies used in our implementation. Does it make sense to use the same shared `Backoff` object for both pushes and pops in our `LockFreeStack<T>` object? How else could we structure the backoff in space and time in the `EliminationBackoffStack<T>`?

Exercise 130. Implement a stack algorithm assuming there is a bound, in any state of the execution, on the total difference between the number of pushes and pops to the stack.

Exercise 131. Consider the problem of implementing a bounded stack using an array indexed by a `top` counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment `top` to reserve an array entry, and then store the item at that index. To pop an item, decrement `top`, and return the item at the previous `top` index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the `top` counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. He decides to adapt the dual-data structure approach of Chapter 10 to implement a *dual* stack. His `DualStack<T>` class splits `push()` and `pop()` methods into *reservation* and *fulfillment* steps. Bob's implementation appears in Fig. 11.10.

The stack's top is indexed by the `top` field, an `AtomicInteger` manipulated only by `getAndIncrement()` and `getAndDecrement()` calls. Bob's `push()`

```
1  public class DualStack<T> {
2    private class Slot {
3      boolean full = false;
4      volatile T value = null;
5    }
6    Slot[] stack;
7    int capacity;
8    private AtomicInteger top = new AtomicInteger(0); // array index
9    public DualStack(int myCapacity) {
10     capacity = myCapacity;
11     stack = (Slot[]) new Object[capacity];
12     for (int i = 0; i < capacity; i++) {
13       stack[i] = new Slot();
14     }
15   }
16   public T pop() throws EmptyException {
17     while (true) {
18       int i = top.getAndDecrement();
19       if (i <= 0) { // is stack empty?
20         throw new EmptyException();
21        } else if (i-1 < capacity){
22          while (!stack [i-1].full ) {};
23          T value = stack[i-1].value;
24          stack[i-1].full = false;
25          return value ; //pop fulfilled
26         }
27      }
28   }
29   public T pop() throws EmptyException {
30     while (true) {
31       int i = top.getAndDecrement();
32       if (i < 0) { // is stack empty?
33         throw new EmptyException();
34       } else if (i < capacity - 1) {
35         while (!stack[i].full){};
36         T value = stack[i].value;
37         stack[i].full = false;
38         return value; //pop fulfilled
39       }
40     }
41   }
42 }
```

Figure 11.10 Bob's problematic dual stack.

method's reservation step reserves a slot by applying `getAndIncrement()` to `top`. Suppose the call returns index i. If i is in the range $0 \ldots$ `capacity` $- 1$, the reservation is complete. In the fulfillment phase, `push`(x) stores x at index i in the array, and raises the `full` flag to indicate that the value is ready to be read. The `value` field must be **`volatile`** to guarantee that once `flag` is raised, the value has already been written to index i of the array.

If the index returned from `push()`'s `getAndIncrement()` is less than 0, the `push()` method repeatedly retries `getAndIncrement()` until it returns an index greater than or equal to 0. The index could be less than 0 due to `getAndDecrement()` calls of failed `pop()` calls to an empty stack. Each such failed `getAndDecrement()` decrements the `top` by one more past the 0 array bound. If the index returned is greater than `capacity` -1, `push()` throws an exception because the stack is full.

The situation is symmetric for `pop()`. It checks that the index is within the bounds and removes an item by applying `getAndDecrement()` to `top`, returning index i. If i is in the range $0 \ldots$ `capacity` $- 1$, the reservation is complete. For the fulfillment phase, `pop()` spins on the `full` flag of array slot i, until it detects that the flag is true, indicating that the `push()` call is successful.

What is wrong with Bob's algorithm? Is this an inherent problem or can you think of a way to fix it?

Exercise 132. In Exercise 97 we ask you to implement the `Rooms` interface, reproduced in Fig. 11.11. The `Rooms` class manages a collection of *rooms*, indexed from 0 to m (where m is a known constant). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. The last thread to leave a room triggers an `onEmpty()` handler, which runs while all rooms are empty.

Fig. 11.12 shows an incorrect concurrent stack implementation.

1. Explain why this stack implementation does not work.
2. Fix it by adding calls to a two-room `Rooms` class: one room for pushing and one for popping.

```
1  public interface Rooms {
2    public interface Handler {
3      void onEmpty();
4    }
5    void enter(int i);
6    boolean exit();
7    public void setExitHandler(int i, Rooms.Handler h) ;
8  }
```

Figure 11.11 The `Rooms` interface.

```
1  public class Stack<T> {
2    private AtomicInteger top;
3    private T[] items;
4    public Stack(int capacity) {
5      top = new AtomicInteger();
6      items = (T[]) new Object[capacity];
7    }
8    public void push(T x) throws FullException {
9      int i = top.getAndIncrement();
10     if (i >= items.length) { // stack is full
11       top.getAndDecrement(); // restore state
12       throw new FullException();
13     }
14     items[i] = x;
15   }
16   public T pop() throws EmptyException {
17     int i = top.getAndDecrement() - 1;
18     if (i < 0) {          // stack is empty
19       top.getAndIncrement(); // restore state
20       throw new EmptyException();
21     }
22     return items[i];
23   }
24 }
```

Figure 11.12 Unsynchronized concurrent stack.

Exercise 133. This exercise is a follow-on to Exercise 132. Instead of having the `push()` method throw `FullException`, exploit the push room's exit handler to resize the array. Remember that no thread can be in any room when an exit handler is running, so (of course) only one exit handler can run at a time.

12 Counting, Sorting, and Distributed Coordination

12.1 Introduction

This chapter shows how some important problems that seem inherently sequential can be made highly parallel by "spreading out" coordination tasks among multiple parties. What does this spreading out buy us?

To answer this question, we need to understand how to measure the performance of a concurrent data structure. There are two measures that come to mind: *latency*, the time it takes an individual method call to complete, and *throughput*, the overall rate at which method calls complete. For example, real-time applications might care more about latency, and databases might care more about throughput.

In Chapter 11 we saw how to apply distributed coordination to the `EliminationBackoffStack` class. Here, we cover several useful patterns for distributed coordination: combining, counting, diffraction, and sampling. Some are deterministic, while others use randomization. We also cover two basic structures underlying these patterns: trees and combinatorial networks. Interestingly, for some data structures based on distributed coordination, high throughput does not necessarily mean low latency.

12.2 Shared Counting

We recall from Chapter 10 that a *pool* is a collection of items that provides `put()` and `get()` methods to insert and remove items (Fig. 10.1). Familiar classes such as stacks and queues can be viewed as pools that provide additional fairness guarantees.

One way to implement a pool is to use coarse-grained locking, perhaps making both `put()` and `get()` **`synchronized`** methods. The problem, of course, is that coarse-grained locking is too heavy-handed, because the lock itself creates both a *sequential bottleneck*, forcing all method calls to synchronize, as well as a *hot spot*,

The Art of Multiprocessor Programming. DOI: 10.1016/B978-0-12-397337-5.00012-5

a source of memory contention. We would prefer to have `Pool` method calls work in parallel, with less synchronization and lower contention.

Let us consider the following alternative. The pool's items reside in a cyclic array, where each array entry contains either an item or *null*. We route threads through two counters. Threads calling `put()` increment one counter to choose an array index into which the new item should be placed. (If that entry is full, the thread waits until it becomes empty.) Similarly, threads calling `get()` increment another counter to choose an array index from which the new item should be removed. (If that entry is empty, the thread waits until it becomes full.)

This approach replaces one bottleneck: the lock, with two: the counters. Naturally, two bottlenecks are better than one (think about that claim for a second). We now explore the idea that shared counters need not be bottlenecks, and can be effectively parallelized. We face two challenges.

1. We must avoid *memory contention*, where too many threads try to access the same memory location, stressing the underlying communication network and cache coherence protocols.
2. We must achieve real parallelism. Is incrementing a counter an inherently sequential operation, or is it possible for n threads to increment a counter faster than it takes one thread to increment a counter n times?

We now look at several ways to build highly parallel counters through data structures that coordinate the distribution of counter indexes.

12.3 Software Combining

Here is a linearizable shared counter class using a pattern called *software combining*. A `CombiningTree` is a binary tree of *nodes*, where each node contains bookkeeping information. The counter's value is stored at the root. Each thread is assigned a leaf, and at most two threads share a leaf, so if there are p physical processors, then there are $p/2$ leaves. To increment the counter, a thread starts at its leaf, and works its way up the tree to the root. If two threads reach a node at approximately the same time, then they *combine* their increments by adding them together. One thread, the *active* thread, propagates their combined increments up the tree, while the other, the *passive* thread, waits for the active thread to complete their combined work. A thread may be active at one level and become passive at a higher level.

For example, suppose threads A and B share a leaf node. They start at the same time, and their increments are combined at their shared leaf. The first one, say, B, actively continues up to the next level, with the mission of adding 2 to the counter value, while the second, A, passively waits for B to return from the root with an acknowledgment that A's increment has occurred. At the next level in the tree, B may combine with another thread C, and advance with the renewed intention of adding 3 to the counter value.

When a thread reaches the root, it adds the sum of its combined increments to the counter's current value. The thread then moves back down the tree, notifying each waiting thread that the increments are now complete.

Combining trees have an inherent disadvantage with respect to locks: each increment has a higher latency, that is, the time it takes an individual method call to complete. With a lock, a `getAndIncrement()` call takes $O(1)$ time, while with a `CombiningTree`, it takes $O(\log p)$ time. Nevertheless, a `CombiningTree` is attractive because it promises far better throughput, that is, the overall rate at which method calls complete. For example, using a queue lock, p `getAndIncrement()` calls complete in $O(p)$ time, at best, while using a `CombiningTree`, under ideal conditions where all threads move up the tree together, p `getAndIncrement()` calls complete in $O(\log p)$ time, an exponential improvement. Of course, the actual performance is often less than ideal, a subject examined in detail later on. Still, the `CombiningTree` class, like other techniques we consider later, is intended to benefit throughput, not latency.

Combining trees are also attractive because they can be adapted to apply any commutative function, not just increment, to the value maintained by the tree.

12.3.1 Overview

Although the idea behind a `CombiningTree` is quite simple, the implementation is not. To keep the overall (simple) structure from being submerged in (not-so-simple) detail, we split the data structure into two classes: the `CombiningTree` class manages navigation within the tree, moving up and down the tree as needed, while the `Node` class manages each visit to a node. As you go through the algorithm's description, it might be a good idea to consult Fig. 12.3 that describes an example `CombiningTree` execution.

This algorithm uses two kinds of synchronization. Short-term synchronization is provided by synchronized methods of the `Node` class. Each method locks the node for the duration of the call to ensure that it can read–write node fields without interference from other threads. The algorithm also requires excluding threads from a node for durations longer than a single method call. Such long-term synchronization is provided by a Boolean `locked` field. When this field is *true*, no other thread is allowed to access the node.

Every tree node has a *combining status*, which defines whether the node is in the early, middle, or late stages of combining concurrent requests.

```
enum CStatus{FIRST, SECOND, RESULT, IDLE, ROOT};
```

These values have the following meanings:

- `IDLE`: This node is not in use.
- `FIRST`: One active thread has visited this node, and will return to check whether another passive thread has left a value with which to combine.
- `SECOND`: A second thread has visited this node and stored a value in the node's `value` field to be combined with the active thread's value, but the combined operation is not yet complete.

- RESULT: Both threads' operations have been combined and completed, and the second thread's result has been stored in the node's `result` field.
- ROOT: This value is a special case to indicate that the node is the root, and must be treated specially.

Fig. 12.1 shows the `Node` class's other fields.

To initialize the `CombiningTree` for p threads, we create a width $w \geqslant p/2$ array of `Node` objects. The root is `node`[0], and for $0 < i < w$, the parent of `node`[i] is `node`[$(i-1)/2$]. The leaf nodes are the last $(w+1)/2$ nodes in the array, where thread i is assigned to leaf $i/2$. The root's initial combining state is `ROOT` and the other nodes combining state is `IDLE`. Fig. 12.2 shows the `CombiningTree` class constructor.

The `CombiningTree`'s `getAndIncrement`() method, shown in Fig. 12.4, has four phases. In the *precombining phase* (Lines 16 through 19), the `CombiningTree` class's `getAndIncrement`() method moves up the tree applying `precombine`() to

```
1  public class Node {
2    enum CStatus{IDLE, FIRST, SECOND, RESULT, ROOT};
3    boolean locked;
4    CStatus cStatus;
5    int firstValue, secondValue;
6    int result;
7    Node parent;
8    public Node() {
9      cStatus = CStatus.ROOT;
10     locked = false;
11   }
12   public Node(Node myParent) {
13     parent = myParent;
14     cStatus = CStatus.IDLE;
15     locked = false;
16   }
17   ...
18 }
```

Figure 12.1 The `Node` class: the constructors and fields.

```
1    public CombiningTree(int width) {
2      Node[] nodes = new Node[width - 1];
3      nodes[0] = new Node();
4      for (int i = 1; i < nodes.length; i++) {
5        nodes[i] = new Node(nodes[(i-1)/2]);
6      }
7      leaf = new Node[(width + 1)/2];
8      for (int i = 0; i < leaf.length; i++) {
9        leaf[i] = nodes[nodes.length - i - 1];
10     }
11   }
```

Figure 12.2 The `CombiningTree` class: constructor.

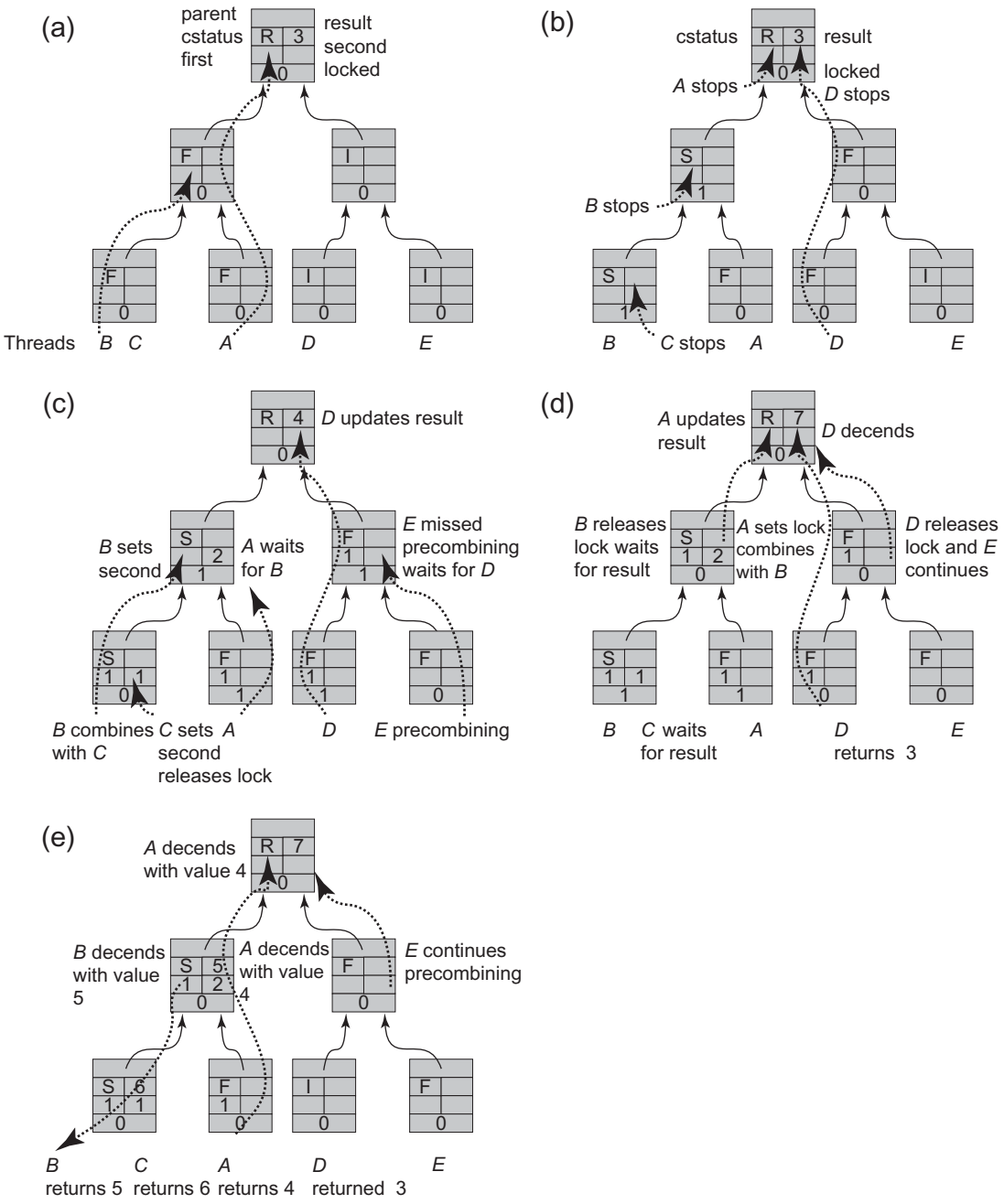

Figure 12.3 The concurrent traversal of a width 8 combining tree by 5 threads. The structure is initialized with all nodes unlocked, the root node having the `CStatus ROOT` and all other nodes having the `CStatus IDLE`.

```
12    public int getAndIncrement() {
13      Stack<Node> stack = new Stack<Node>();
14      Node myLeaf = leaf[ThreadID.get()/2];
15      Node node = myLeaf;
16      // precombining phase
17      while (node.precombine()) {
18        node = node.parent;
19      }
20      Node stop = node;
21      // combining phase
22      node = myLeaf;
23      int combined = 1;
24      while (node != stop) {
25        combined = node.combine(combined);
26        stack.push(node);
27        node = node.parent;
28      }
29      // operation phase
30      int prior = stop.op(combined);
31      // distribution phase
32      while (!stack.empty()) {
33        node = stack.pop();
34        node.distribute(prior);
35      }
36      return prior;
37    }
```

Figure 12.4 The `CombiningTree` class: the `getAndIncrement()` method.

each node. The `precombine()` method returns a Boolean indicating whether the thread was the first to arrive at the node. If so, the `getAndIncrement()` method continues moving up the tree. The `stop` variable is set to the last node visited, which is either the last node at which the thread arrived second, or the root. For example, Part (a) of Fig. 12.3 shows a precombining phase example. Thread ***A***, which is fastest, stops at the root, while ***B*** stops in the middle-level node where it arrived after ***A***, and ***C*** stops at the leaf where it arrived after ***B***.

Fig. 12.5 shows the `Node`'s `precombine()` method. In Line 20, the thread waits until the locked field is false. In Line 21, it tests the combining status.

IDLE

The thread sets the node's status to `FIRST` to indicate that it will return to look for a value for combining. If it finds such a value, it proceeds as the active thread, and the thread that provided that value is passive. The call then returns *true*, instructing the thread to move up the tree.

FIRST

An earlier thread has recently visited this node, and will return to look for a value to combine. The thread instructs the thread to stop moving up the tree (by

```
19   synchronized boolean precombine() {
20     while (locked) wait();
21     switch (cStatus) {
22       case IDLE:
23         cStatus = CStatus.FIRST;
24         return true;
25       case FIRST:
26         locked = true;
27         cStatus = CStatus.SECOND;
28         return false;
29       case ROOT:
30         return false;
31       default:
32         throw new PanicException("unexpected Node state" + cStatus);
33     }
34   }
```

Figure 12.5 The Node class: the precombining phase.

returning *false*), and to start the next phase, computing the value to combine. Before it returns, the thread places a long-term lock on the node (by setting `locked` to *true*) to prevent the earlier visiting thread from proceeding without combining with the thread's value.

ROOT

If the thread has reached the root node, it instructs the thread to start the next phase.
Line 31 is a *default* case that is executed only if an unexpected status is encountered.

> *Pragma* 12.3.1. It is good programming practice always to provide an arm for every possible enumeration value, even if we know it cannot happen. If we are wrong, the program is easier to debug, and if we are right, the program may later be changed even by someone who does not know as much as we do. Always program defensively.

In the *combining phase*, (Fig. 12.4, Lines 21–28), the thread revisits the nodes it visited in the precombining phase, combining its value with values left by other threads. It stops when it arrives at the node `stop` where the precombining phase ended. Later on, we traverse these nodes in reverse order, so as we go we push the nodes we visit onto a stack.

The `Node` class's `combine()` method, shown in Fig. 12.6, adds any values left by a recently arrived passive process to the values combined so far. As before, the thread first waits until the `locked` field is *false*. It then sets a long-term lock on the node, to ensure that late-arriving threads do not expect to combine with it. If the status is `SECOND`, it adds the other thread's value to the accumulated value, otherwise it returns the value unchanged. In Part (c) of Fig. 12.3, thread *A* starts

```
35    synchronized int combine(int combined) {
36      while (locked) wait();
37      locked = true;
38      firstValue = combined;
39      switch (cStatus) {
40        case FIRST:
41          return firstValue;
42        case SECOND:
43          return firstValue + secondValue;
44        default:
45          throw new PanicException("unexpected Node state " + cStatus);
46      }
47    }
```

Figure 12.6 The `Node` class: the combining phase. This method applies addition to `FirstValue` and `SecondValue`, but any other commutative operation would work just as well.

```
48    synchronized int op(int combined) {
49      switch (cStatus) {
50        case ROOT:
51          int prior = result;
52          result += combined;
53          return prior;
54        case SECOND:
55          secondValue = combined;
56          locked = false;
57           notifyAll(); // wake up waiting threads
58          while (cStatus != CStatus.RESULT) wait();
59          locked = false;
60           notifyAll();
61          cStatus = CStatus.IDLE;
62          return result;
63        default:
64          throw new PanicException("unexpected Node state");
65      }
66    }
```

Figure 12.7 The `Node` class: applying the operation.

ascending the tree in the combining phase. It reaches the second level node locked by thread *B* and waits. In Part (d), *B* releases the lock on the second level node, and *A*, seeing that the node is in a `SECOND` combining state, locks the node and moves to the root with the combined value 3, the sum of the `FirstValue` and `SecondValue` fields written respectively by *A* and *B*.

At the start of the operation phase (Lines 29 and 30), the thread has now combined all method calls from lower-level nodes, and now examines the node where it stopped at the end of the precombining phase (Fig. 12.7). If the node is the root, as in Part (d) of Fig. 12.3, then the thread, in this case *A*, carries out the combined `getAndIncrement()` operations: it adds its accumulated value (3 in the example) to the `result` and returns the `prior` value. Otherwise, the thread unlocks the node, notifies any blocked thread, deposits its value as the

```
67   synchronized void distribute(int prior) {
68     switch (cStatus) {
69       case FIRST:
70         cStatus = CStatus.IDLE;
71         locked = false;
72         break;
73       case SECOND:
74         result = prior + firstValue;
75         cStatus = CStatus.RESULT;
76         break;
77       default:
78         throw new PanicException("unexpected Node state");
79     }
80     notifyAll();
81   }
```

Figure 12.8 The Node class: the distribution phase.

SecondValue, and waits for the other thread to return a result after propagating the combined operations toward the root. For example, this is the sequence of actions taken by thread *B* in Parts (c) and (d) of Fig. 12.3.

When the result arrives, *A* enters the *distribution phase*, propagating the result down the tree. In this phase (Lines 31–36), the thread moves down the tree, releasing locks, and informing passive partners of the values they should report to their own passive partners, or to the caller (at the lowest level). The distribute method is shown in Fig. 12.8. If the state of the node is FIRST, no thread combines with the distributing thread, and it can reset the node to its initial state by releasing the lock and setting the state to IDLE. If, on the other hand, the state is SECOND, the distributing thread updates the result to be the sum of the prior value brought from higher up the tree, and the FIRST value. This reflects a situation in which the active thread at the node managed to perform its increment before the passive one. The passive thread waiting to get a value reads the result once the distributing thread sets the status to RESULT. For example, in Part (e) of Fig. 12.3, the active thread *A* executes its distribution phase in the middle level node, setting the result to 5, changing the state to RESULT, and descending down to the leaf, returning the value 4 as its output. The passive thread *B* awakes and sees that the middle-level node's state has changed, and reads result 5.

12.3.2 An Extended Example

Fig. 12.3 describes the various phases of a CombiningTree execution. There are five threads labeled *A* through *E*. Each node has six fields, as shown in Fig. 12.1. Initially, all nodes are unlocked and all but the root are in an IDLE combining state. The counter value in the initial state in Part (a) is 3, the result of an earlier computation. In Part (a), to perform a getAndIncrement(), threads *A* and *B* start the precombining phase. *A* ascends the tree, changing the nodes it visits from IDLE to FIRST, indicating that it will be the active thread in combining the values up the tree. Thread *B* is the active thread at its leaf node, but has not

yet arrived at the second-level node shared with ***A***. In Part (b), ***B*** arrives at the second-level node and stops, changing it from `FIRST` to `SECOND`, indicating that it will collect its combined values and wait here for ***A*** to proceed with them to the root. ***B*** locks the node (changing the `locked` field from *false* to *true*), preventing ***A*** from proceeding with the combining phase without ***B***'s combined value. But ***B*** has not combined the values. Before it does so, ***C*** starts precombining, arrives at the leaf node, stops, and changes its state to `SECOND`. It also locks the node to prevent ***B*** from ascending without its input to the combining phase. Similarly, ***D*** starts precombining and successfully reaches the root node. Neither ***A*** nor ***D*** changes the root node state, and in fact it never changes. They simply mark it as the node where they stopped precombining. In Part (c) ***A*** starts up the tree in the combining phase. It locks the leaf so that any later thread will not be able to proceed in its precombining phase, and will wait until ***A*** completes its combining and distribution phases. It reaches the second-level node, locked by ***B***, and waits. In the meantime, ***C*** starts combining, but since it stopped at the leaf node, it executes the `op()` method on this node, setting `SecondValue` to 1 and then releasing the lock. When ***B*** starts its combining phase, the leaf node is unlocked and marked `SECOND`, so ***B*** writes 1 to `FirstValue` and ascends to the second-level node with a combined value of 2, the result of adding the `FirstValue` and `SecondValue` fields.

When it reaches the second level node, the one at which it stopped in the precombining phase, it calls the `op()` method on this node, setting `SecondValue` to 2. ***A*** must wait until it releases the lock. Meanwhile, in the right-hand side of the tree, ***D*** executes its combining phase, locking nodes as it ascends. Because it meets no other threads with which to combine, it reads 3 in the `result` field in the root and updates it to 4. Thread ***E*** then starts precombining, but is late in meeting ***D***. It cannot continue precombining as long as ***D*** locks the second-level node. In Part (d), ***B*** releases the lock on the second-level node, and ***A***, seeing that the node is in state `SECOND`, locks the node and moves to the root with the combined value 3, the sum of the `FirstValue` and `SecondValue` fields written, respectively, by ***A*** and ***B***. ***A*** is delayed while ***D*** completes updating the root. Once ***D*** is done, ***A*** reads 4 in the root's `result` field and updates it to 7. ***D*** descends the tree (by popping its local `Stack`), releasing the locks and returning the value 3 that it originally read in the root's `result` field. ***E*** now continues its ascent in the precombining phase. Finally, in Part (e), ***A*** executes its distribution phase. It returns to the middle-level node, setting `result` to 5, changing the state to `RESULT`, and descending to the leaf, returning the value 4 as its output. ***B*** awakens and sees the state of the middle-level node has changed, reads 5 as the `result`, and descends to its leaf where it sets the `result` field to 6 and the state to `RESULT`. ***B*** then returns 5 as its output. Finally, ***C*** awakens and observes that the leaf node state has changed, reads 6 as the `result`, which it returns as its output value. Threads ***A*** through ***D*** return values 3 to 6 which fit the root's `result` field value of 7. The linearization order of the `getAndIncrement()` method calls by the different threads is determined by their order in the tree during the precombining phase.

12.3.3 Performance and Robustness

Like all the algorithms described in this chapter, `CombiningTree` throughput depends in complex ways on the characteristics both of the application and of the underlying architecture. Nevertheless, it is worthwhile to review, in qualitative terms, some experimental results from the literature. Readers interested in detailed experimental results (mostly for obsolete architectures) may consult the chapter notes.

As a thought experiment, a `CombiningTree` should provide high throughput under ideal circumstances when each thread can combine its increment with another's. But it may provide poor throughput under worst-case circumstances, where many threads arrive late at a locked node, missing the chance to combine, and are forced to wait for the earlier request to ascend and descend the tree.

In practice, experimental evidence supports this informal analysis. The higher the contention, the greater the observed rate of combining, and the greater the observed speed-up. Worse is better. Combining trees are less attractive when concurrency is low. The combining rate decreases rapidly as the arrival rate of increment requests is reduced. Throughput is sensitive to the arrival rate of requests.

Because combining increases throughput, and failure to combine does not, it makes sense for a request arriving at a node to wait for a reasonable duration for another thread to arrive with a increment with which to combine. Not surprisingly, it makes sense to wait for a short time when the contention is low, and longer when contention is high. When contention is sufficiently high, unbounded waiting works very well.

An algorithm is *robust* if it performs well in the presence of large fluctuations in request arrival times. The literature suggests that the `CombiningTree` algorithm with a fixed waiting time is not robust, because high variance in request arrival rates seems to reduce the combining rate.

12.4 Quiescently Consistent Pools and Counters

> First shalt thou take out the Holy Pin. Then shalt thou count to three, no more, no less. Three shall be the number thou shalt count, and the number of the counting shall be three. . . . Once the number three, being the third number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thy foe, who, being naughty in my sight, shall snuff it.
>
> From *Monty Python and the Holy Grail.*

Not all applications require linearizable counting. Indeed, counter-based `Pool` implementations require only quiescently consistent[1] counting: all that matters is that the counters produce no duplicates and no omissions. It is enough that

1 See Chapter 3 for a detailed definition of quiescent consistency.

for every item placed by a `put()` in an array entry, another thread eventually executes a `get()` that accesses that entry, eventually matching `put()` and `get()` calls. (Wrap-around may still cause multiple `put()` calls or `get()` calls to compete for the same array entry.)

12.5 Counting Networks

Students of Tango know that the partners must be tightly coordinated: if they do not move together, the dance does not work, no matter how skilled the dancers may be as individuals. In the same way, combining trees must be tightly coordinated: if requests do not arrive together, the algorithm does not work efficiently, no matter how fast the individual processes.

In this chapter, we consider *counting networks*, which look less like Tango and more like a Rave: each participant moves at its own pace, but collectively the counter delivers a quiescently consistent set of indexes with high throughput.

Let us imagine that we replace the combining tree's single counter with multiple counters, each of which distributes a subset of indexes (see Fig. 12.9). We allocate w counters (in the figure $w = 4$), each of which distributes a set of unique indexes modulo w (in the figure, for example, the second counter distributes 2, 6, 10, ... $i \cdot w + 2$ for increasing i). The challenge is how to distribute the threads among the counters so that there are no duplications or omissions, and how to do so in a distributed and loosely coordinated style.

12.5.1 Networks That Count

A *balancer* is a simple switch with two input wires and two output wires, called the *top* and *bottom* wires (or sometimes the *north* and *south* wires). Tokens arrive on the balancer's input wires at arbitrary times, and emerge on their output wires, at some later time. A balancer can be viewed as a toggle: given a stream of input tokens, it sends one token to the top output wire, and the next to the bottom, and so on, effectively balancing the number of tokens between the two wires (see Fig. 12.10). More precisely, a balancer has two states: *up* and *down*. If

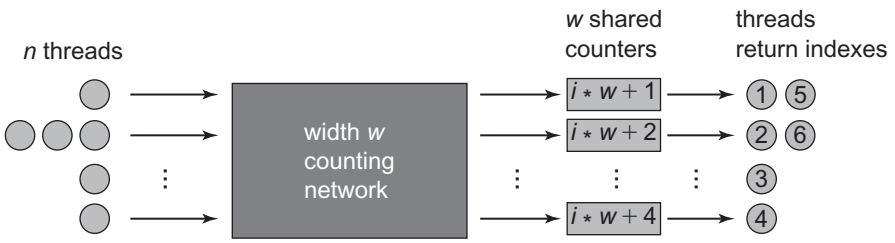

Figure 12.9 A quiescently consistent shared counter based on $w = 4$ counters preceded by a counting network. Threads traverse the counting network to choose which counters to access.

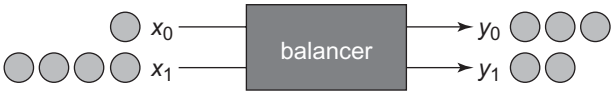

Figure 12.10 A balancer. Tokens arrive at arbitrary times on arbitrary input lines and are redirected to ensure that when all tokens have exited the balancer, there is at most one more token on the top wire than on the bottom one.

the state is *up*, the next token exits on the top wire, otherwise it exits on the bottom wire.

We use x_0 and x_1 to denote the number of tokens that respectively arrive on a balancer's top and bottom input wires, and y_0 and y_1 to denote the number that exit on the top and bottom output wires. A balancer never creates tokens: at all times.

$$x_0 + x_1 \geqslant y_0 + y_1.$$

A balancer is said to be *quiescent* if every token that arrived on an input wire has emerged on an output wire:

$$x_0 + x_1 = y_0 + y_1.$$

A *balancing network* is constructed by connecting some balancers' output wires to other balancers' input wires. A balancing network of width w has input wires $x_0, x_1, \ldots, x_{w-1}$ (not connected to output wires of balancers), and w output wires $y_0, y_1, \ldots, y_{w-1}$ (similarly unconnected). The balancing network's *depth* is the maximum number of balancers one can traverse starting from any input wire. We consider only balancing networks of finite depth (meaning the wires do not form a loop). Like balancers, balancing networks do not create tokens:

$$\sum x_i \geqslant \sum y_i.$$

(We usually drop indexes from summations when we sum over every element in a sequence.) A balancing network is *quiescent* if every token that arrived on an input wire has emerged on an output wire:

$$\sum x_i = \sum y_i.$$

So far, we have described balancing networks as if they were switches in a network. On a shared-memory multiprocessor, however, a balancing network can be implemented as an object in memory. Each balancer is an object, whose wires are references from one balancer to another. Each thread repeatedly traverses the object, starting on some input wire, and emerging at some output wire, effectively shepherding a token through the network.

Some balancing networks have interesting properties. The network shown in Fig. 12.11 has four input wires and four output wires. Initially, all balancers are *up*. We can check for ourselves that if any number of tokens enter the network, in any order, on any set of input wires, then they emerge in a regular pattern on the output wires. Informally, no matter how token arrivals are distributed among the

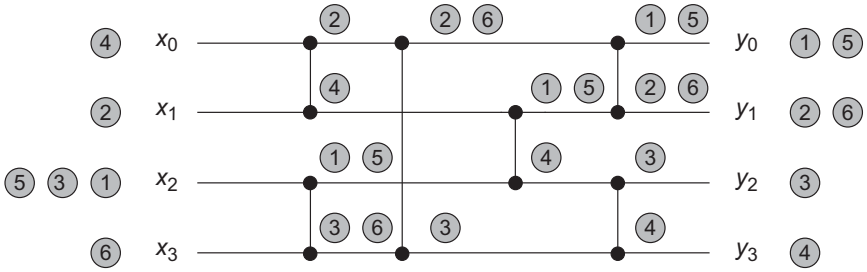

Figure 12.11 A sequential execution of a BITONIC [4] counting network. Each vertical line represents a balancer, and each balancer's two input and output wires are the horizontal lines it connects to at the dots. In this sequential execution, tokens pass through the network, one completely after the other in the order specified by the numbers on the tokens. We track every token as it passes through the balancers on the way to an output wire. For example, token number 3 enters on wire 2, goes down to wire 3, and ends up on wire 2. Notice how the step property is maintained in every balancer, and also in the network as a whole.

input wires, the output distribution is balanced across the output wires, where the top output wires are filled first. If the number of tokens n is a multiple of four (the network width), then the same number of tokens emerges from each wire. If there is one excess token, it emerges on output wire 0, if there are two, they emerge on output wires 0 and 1, and so on. In general,
if

$$n = \sum x_i$$

then

$$y_i = (n/w) + (i \bmod w).$$

We call this property the *step property*.

Any balancing network that satisfies the step property is called a *counting network*, because it can easily be adapted to count the number of tokens that have traversed the network. Counting is done, as we described earlier in Fig. 12.9, by adding a local counter to each output wire i, so that tokens emerging on that wire are assigned consecutive numbers $i, i + w, \ldots, i + (y_i - 1)w$.

The step property can be defined in a number of ways which we use interchangeably.

Lemma 12.5.1. If $y_0, \ldots, y_{w-1}$ is a sequence of nonnegative integers, the following statements are all equivalent:

1. For any $i < j$, $0 \leqslant y_i - y_j \leqslant 1$.
2. Either $y_i = y_j$ for all i, j, or there exists some c such that for any $i < c$ and $j \geqslant c$, $y_i - y_j = 1$.
3. If $m = \sum y_i$, $y_i = \left\lceil \frac{m-i}{w} \right\rceil$.

12.5.2 The Bitonic Counting Network

In this section we describe how to generalize the counting network of Fig. 12.11 to a counting network whose width is any power of 2. We give an inductive construction.

When describing counting networks, we do not care about when tokens arrive, we care only that when the network is quiescent, the number of tokens exiting on the output wires satisfies the step property. Define a width w *sequence* of inputs or outputs $x = x_0, \ldots, x_{w-1}$ to be a collection of tokens, partitioned into w subsets x_i. The x_i are the input tokens that arrive or leave on wire i.

We define the width-$2k$ balancing network MERGER $[2k]$ as follows. It has two input sequences of width k, x and x', and a single output sequence y of width $2k$. In any quiescent state, if x and x' both have the step property, then so does y. The MERGER $[2k]$ network is defined inductively (see Fig. 12.12). When k is equal to 1, the MERGER $[2k]$ network is a single balancer. For $k > 1$, we construct the MERGER $[2k]$ network with input sequences x and x' from two MERGER $[k]$ networks and k balancers. Using a MERGER $[k]$ network, we merge the even subsequence $x_0, x_2, \ldots, x_{k-2}$ of x with the odd subsequence $x'_1, x'_3, \ldots, x'_{k-1}$ of x' (that is, the sequence $x_0, \ldots, x_{k-2}, x'_1, \ldots, x'_{k-1}$ is the input to the MERGER $[k]$ network), while with a second MERGER $[k]$ network we merge the odd subsequence of x with the even subsequence of x'. We call the outputs of these two MERGER $[k]$ networks z and z'. The final stage of the network combines z and z' by sending each pair of wires z_i and z'_i into a balancer whose outputs yield y_{2i} and y_{2i+1}.

The MERGER $[2k]$ network consists of $\log 2k$ layers of k balancers each. It provides the step property for its outputs only when its two input sequences also have the step property, which we ensure by filtering the inputs through smaller balancing networks.

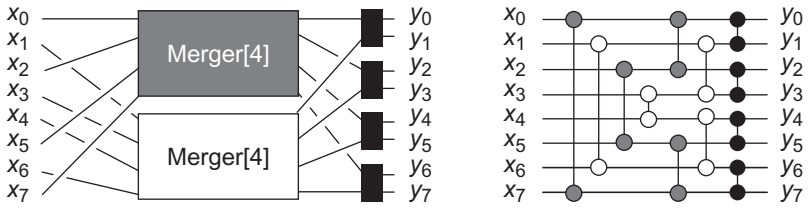

Figure 12.12 On the left-hand side we see the logical structure of a MERGER [8] network, into which feed two BITONIC [4] networks, as depicted in Fig. 12.11. The gray MERGER [4] network has as inputs the even wires coming out of the top BITONIC [4] network, and the odd ones from the lower BITONIC [4] network. In the lower MERGER [4] the situation is reversed. Once the wires exit the two MERGER [4] networks, each pair of identically numbered wires is combined by a balancer. On the right-hand side we see the physical layout of a MERGER [8] network. The different balancers are color coded to match the logical structure in the left-hand figure.

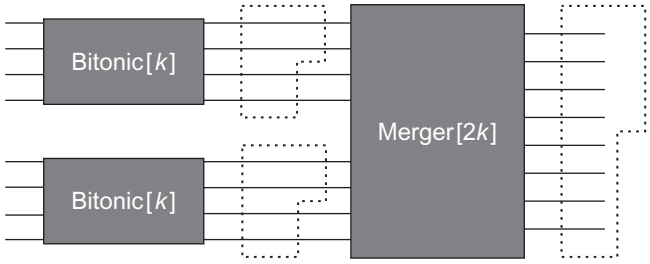

Figure 12.13 The recursive structure of a BITONIC [2k] Counting Network. Two BITONIC [k] counting networks feed into a MERGER [2k] balancing network.

The BITONIC $[2k]$ network is constructed by passing the outputs from two BITONIC $[k]$ networks into a MERGER $[2k]$ network, where the induction is grounded in the BITONIC [2] network consisting of a single balancer, as depicted in Fig. 12.13. This construction gives us a network consisting of $\left(\begin{smallmatrix}\log 2k+1\\ 2\end{smallmatrix}\right)$ layers each consisting of k balancers.

A Software Bitonic Counting Network

So far, we have described counting networks as if they were switches in a network. On a shared-memory multiprocessor however, a balancing network can be implemented as an object in memory. Each balancer is an object, whose wires are references from one balancer to another. Each thread repeatedly traverses the object, starting on some input wire and emerging at some output wire, effectively shepherding a token through the network. Here, we show how to implement a BITONIC [2] network as a shared-memory data structure.

The `Balancer` class (Fig. 12.14) has a single Boolean field: `toggle`. The synchronized `traverse()` method complements the `toggle` field and returns as output wire, either 0 or 1. The `Balancer` class's `traverse()` method does not need an argument because the wire on which a token exits a balancer does not depend on the wire on which it enters.

The `Merger` class (Fig. 12.15) has three fields: the `width` field must be a power of 2, `half[]` is a two-element array of half-width `Merger` objects (empty if the network has width 2), and `layer[]` is an array of `width/2` balancers implementing the final network layer.

The class provides a `traverse(`i`)` method, where i is the wire on which the token enters. (For merger networks, unlike balancers, a token's path depends on its input wire.) If the token entered on the lower `width`/2 wires, then it passes through `half[0]`, otherwise `half[1]`. No matter which half-width merger network it traverses, a balancer that emerges on wire i is fed to the i^{th} balancer at `layer[`i`]`.

The `Bitonic` class (Fig. 12.16) also has three fields: `width` is the width (a power of 2), `half[]` is a two-element array of half-width `Bitonic[]` objects,

```
1  public class Balancer {
2    boolean toggle = true;
3    public synchronized int traverse() {
4      try {
5        if (toggle) {
6          return 0;
7        } else {
8          return 1;
9        }
10     } finally {
11       toggle = !toggle;
12     }
13   }
14 }
```

Figure 12.14 The `Balancer` class: a **`synchronized`** implementation.

```
1  public class Merger {
2    Merger[] half;  // two half-width merger networks
3    Balancer[] layer; // final layer
4    final int width;
5    public Merger(int myWidth) {
6      width = myWidth;
7      layer = new Balancer[width / 2];
8      for (int i = 0; i < width / 2; i++) {
9        layer[i] = new Balancer();
10     }
11     if (width > 2) {
12       half = new Merger[]{new Merger(width/2), new Merger(width/2)};
13     }
14   }
15   public int traverse(int input) {
16     int output = 0;
17     if (input < width / 2) {
18       output = half[input % 2].traverse(input / 2);
19     } else {
20       output = half[1 - (input % 2)].traverse(input / 2);
21     return (2 * output) + layer[output].traverse();
22   }
23 }
```

Figure 12.15 The `Merger` class.

and `merger` is a full width `Merger` network `width`. If the network has width 2, the `half[]` array is uninitialized. Otherwise, each element of `half[]` is initialized to a half-width `Bitonic[]` network.

The class provides a `traverse(`*i*`)` method. If the token entered on the lower `width`/2 wires, then it passes through `half`[0], otherwise `half`[1]. A token that

```
1  public class Bitonic {
2    Bitonic[] half; // two half-width bitonic networks
3    Merger merger; // final merger layer
4    final int width; // network width
5    public Bitonic(int myWidth) {
6      width = myWidth;
7      merger = new Merger(width);
8      if (width > 2) {
9        half = new Bitonic[]{new Bitonic(width/2), new Bitonic(width/2)};
10     }
11   }
12   public int traverse(int input) {
13     int output = 0;
14     if (width > 2) {
15       output = half[input / (width / 2)].traverse(input / 2);
16     }
17     return merger.traverse((input >= (size / 2) ? (size / 2) : 0) + output);
18   }
19 }
```

Figure 12.16 The `Bitonic[]` class.

emerges from the half-merger subnetwork on wire i then traverses the final merger network from input wire i.

Notice that this class uses a simple synchronized `Balancer` implementation, but that if the `Balancer` implementation were lock-free (or wait-free) the network implementation as a whole would be lock-free (or wait-free).

Proof of Correctness

We now show that BITONIC $[w]$ is a counting network. The proof proceeds as a progression of arguments about the token sequences passing through the network. Before examining the network itself, here are some simple lemmas about sequences with the step property.

Lemma 12.5.2. If a sequence has the step property, then so do all its subsequences.

Lemma 12.5.3. If $x_0, \ldots, x_{k-1}$ has the step property, then its even and odd subsequences satisfy:

$$\sum_{i=0}^{k/2-1} x_{2i} = \left\lceil \sum_{i=0}^{k-1} x_i/2 \right\rceil \qquad \text{and} \qquad \sum_{i=0}^{k/2-1} x_{2i+1} = \left\lfloor \sum_{i=0}^{k-1} x_i/2 \right\rfloor.$$

Proof: Either $x_{2i} = x_{2i+1}$ for $0 \leqslant i < k/2$, or by Lemma 12.5.1, there exists a unique j such that $x_{2j} = x_{2j+1} + 1$ and $x_{2i} = x_{2i+1}$ for all $i \neq j$, $0 \leqslant i < k/2$. In the first case, $\sum x_{2i} = \sum x_{2i+1} = \sum x_i/2$, and in the second case $\sum x_{2i} = \lceil \sum x_i/2 \rceil$ and $\sum x_{2i+1} = \lfloor \sum x_i/2 \rfloor$. □

Lemma 12.5.4. Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum x_i = \sum y_i$, then $x_i = y_i$ for all $0 \leqslant i < k$.

Proof: Let $m = \sum x_i = \sum y_i$. By Lemma 12.5.1, $x_i = y_i = \left\lceil \frac{m-i}{k} \right\rceil$. □

Lemma 12.5.5. Let $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ be arbitrary sequences having the step property. If $\sum x_i = \sum y_i + 1$, then there exists a unique j, $0 \leqslant j < k$, such that $x_j = y_j + 1$, and $x_i = y_i$ for $i \neq j$, $0 \leqslant i < k$.

Proof: Let $m = \sum x_i = \sum y_i + 1$. By Lemma 12.5.1, $x_i = \left\lceil \frac{m-1}{k} \right\rceil$ and $y_i = \left\lceil \frac{m-1-i}{k} \right\rceil$. These two terms agree for all i, $0 \leqslant i < k$, except for the unique i such that $i = m - 1 \pmod{k}$. □

We now show that the MERGER $[w]$ network preserves the step property.

Lemma 12.5.6. If MERGER $[2k]$ is quiescent, and its inputs $x_0, \ldots, x_{k-1}$ and $x'_0, \ldots, x'_{k-1}$ both have the step property, then its outputs $y_0, \ldots, y_{2k-1}$ also have the step property.

Proof: We argue by induction on $\log k$. It may be worthwhile to consult Fig. 12.17 which shows an example of the proof structure for a MERGER [8] network.

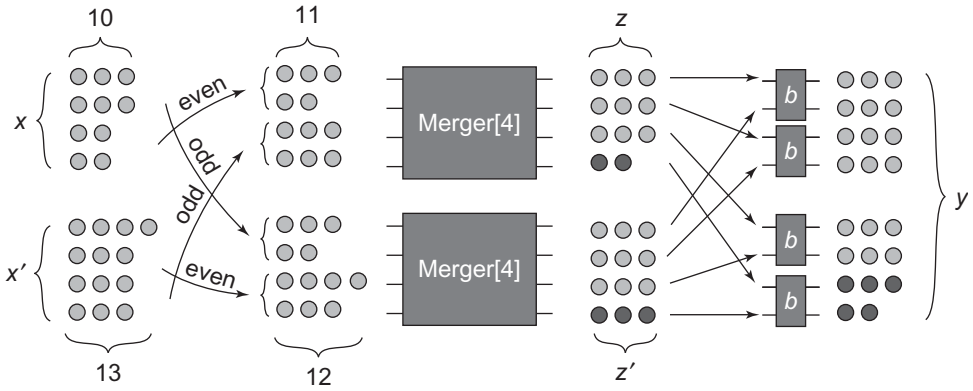

Figure 12.17 The inductive proof that a MERGER [8] network correctly merges two width 4 sequences *x* and *x*' that have the step property into a single width 8 sequence *y* that has the step property. The odd and even width 2 subsequences of *x* and *x*' all have the step property. Moreover, the difference in the number of tokens between the even sequence from one and the odd sequence from the other is at most 1 (in this example, 11 and 12 tokens, respectively). It follows from the induction hypothesis that the outputs *z* and *z*' of the two MERGER [4] networks have the step property, with at most 1 extra token in one of them. This extra token must fall on a specific numbered wire (wire 3 in this case) leading into the same balancer. In this figure, these tokens are darkened. They are passed to the southern-most balancer, and the extra token is pushed north, ensuring the final output has the step property.

If $2k = 2$, MERGER $[2k]$ is just a balancer, and its outputs are guaranteed to have the step property by the definition of a balancer.

If $2k > 2$, let $z_0, \ldots, z_{k-1}$ be the outputs of the first MERGER $[k]$ subnetwork which merges the even subsequence of x with the odd subsequence of x'. Let $z'_0, \ldots, z'_{k-1}$ be the outputs of the second MERGER $[k]$ subnetwork. Since x and x' have the step property by assumption, so do their even and odd subsequences (Lemma 12.5.2), and hence so do z and z' (induction hypothesis). Furthermore, $\sum z_i = \lceil \sum x_i/2 \rceil + \lfloor \sum x'_i/2 \rfloor$ and $\sum z'_i = \lfloor \sum x_i/2 \rfloor + \lceil \sum x'_i/2 \rceil$ (Lemma 12.5.3). A straightforward case analysis shows that $\sum z_i$ and $\sum z'_i$ can differ by at most 1.

We claim that $0 \leqslant y_i - y_j \leqslant 1$ for any $i < j$. If $\sum z_i = \sum z'_i$, then Lemma 12.5.4 implies that $z_i = z'_i$ for $0 \leqslant i < k/2$. After the final layer of balancers,

$$y_i - y_j = z_{\lfloor i/2 \rfloor} - z_{\lfloor j/2 \rfloor},$$

and the result follows because z has the step property.

Similarly, if $\sum z_i$ and $\sum z'_i$ differ by one, Lemma 12.5.5 implies that $z_i = z'_i$ for $0 \leqslant i < k/2$, except for a unique ℓ such that z_ℓ and z'_ℓ differ by one. Let $\max(z_\ell, z'_\ell) = x + 1$ and $\min(z_\ell, z'_\ell) = x$ for some nonnegative integer x. From the step property for z and z' we have, for all $i < \ell$, $z_i = z'_i = x + 1$ and for all $i > \ell$ $z_i = z'_i = x$. Since z_ℓ and z'_ℓ are joined by a balancer with outputs $y_{2\ell}$ and $y_{2\ell+1}$, it follows that $y_{2\ell} = x + 1$ and $y_{2\ell+1} = x$. Similarly, z_i and z'_i for $i \neq \ell$ are joined by the same balancer. Thus, for any $i < \ell$, $y_{2i} = y_{2i+1} = x + 1$ and for any $i > \ell$, $y_{2i} = y_{2i+1} = x$. The step property follows by choosing $c = 2\ell + 1$ and applying Lemma 12.5.1. □

The proof of the following theorem is now immediate.

***Theorem* 12.5.1.** In any quiescent state, the outputs of BITONIC $[w]$ have the step property.

A Periodic Counting Network

In this section, we show that the Bitonic network is not the only counting network with depth $O(log^2 w)$. We introduce a new counting network with the remarkable property that it is *periodic*, consisting of a sequence of identical subnetworks, as depicted in Fig. 12.18. We define the network BLOCK $[k]$ as follows. When k is equal to 2, the BLOCK $[k]$ network consists of a single balancer. The BLOCK $[2k]$ network for larger k is constructed recursively. We start with two BLOCK $[k]$ networks A and B. Given an input sequence x, the input to A is x^A, and the input to B is x^B. Let y be the output sequence for the two subnetworks, where y^A is the output sequence for A and y^B the output sequence for B. The final stage of the network combines each y^A_i and y^B_i in a single balancer, yielding final outputs z_{2i} and z_{2i+1}.

Fig. 12.19 describes the recursive construction of a BLOCK [8] network. The PERIODIC $[2k]$ network consists of $\log k$ BLOCK $[2k]$ networks joined so that

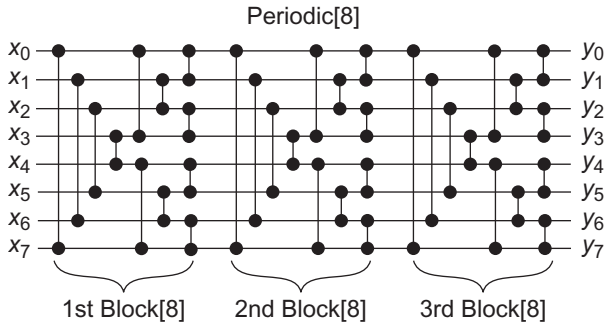

Figure 12.18 A PERIODIC [8] counting network constructed from 3 identical BLOCK [8] networks.

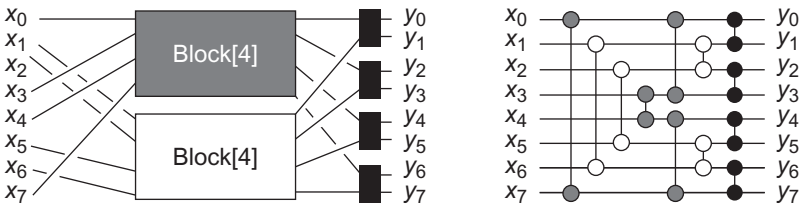

Figure 12.19 The left-hand side illustrates a BLOCK [8] network, into which feed two PERIODIC [4] networks. The right-hand illustrates the physical layout of a MERGER [8] network. The balancers are color-coded to match the logical structure in the left-hand figure.

the i^{th} output wire of one is the i^{th} wire of the next. Fig. 12.18 is a PERIODIC [8] counting network.[2]

A Software Periodic Counting Network

Here is how to implement the Periodic network in software. We reuse the `Balancer` class in Fig. 12.14. A single layer of a BLOCK [w] network is implemented by the LAYER [w] network (Fig. 12.20). A LAYER [w] network joins input wires i and $w - i - 1$ to the same balancer.

In the BLOCK [w] class (Fig. 12.21), after the token emerges from the initial LAYER [w] network, it passes through one of two half-width BLOCK [$w/2$] networks (called *north* and *south*).

The PERIODIC [w] network (Fig. 12.22) is implemented as an array of $\log w$ BLOCK [w] networks. Each token traverses each block in sequence, where the output wire taken on each block is the input wire for its successor. (The chapter notes cite the proof that the PERIODIC [w] is a counting network.)

2 While the BLOCK [$2k$] and MERGER [$2k$] networks may look the same, they are not: there is no permutation of wires that yields one from the other.

```
1  public class Layer {
2    int width;
3    Balancer[] layer;
4    public Layer(int width) {
5      this.width = width;
6      layer = new Balancer[width];
7      for (int i = 0; i < width / 2; i++) {
8        layer[i] = layer[width-i-1] = new Balancer();
9      }
10   }
11   public int traverse(int input) {
12     int toggle = layer[input].traverse();
13     int hi, lo;
14     if (input < width / 2) {
15       lo = input;
16       hi = width - input - 1;
17     } else {
18       lo = width - input - 1;
19       hi = input;
20     }
21     if (toggle == 0) {
22       return lo;
23     } else {
24       return hi;
25     }
26   }
27 }
```

Figure 12.20 The Layer network.

12.5.3 Performance and Pipelining

How does counting network throughput vary as a function of the number of threads and the network width? For a fixed network width, throughput rises with the number of threads up to a point, and then the network *saturates*, and throughput remains constant or declines. To understand these results, let us think of a counting network as a pipeline.

- If the number of tokens concurrently traversing the network is less than the number of balancers, then the pipeline is partly empty, and throughput suffers.
- If the number of concurrent tokens is greater than the number of balancers, then the pipeline becomes clogged because too many tokens arrive at each balancer at the same time, resulting in per-balancer contention.
- Throughput is maximized when the number of tokens is roughly equal to the number of balancers.

If an application needs a counting network, then the best size network to choose is one that ensures that the number of tokens traversing the balancer at any time is roughly equal to the number of balancers.

```
1  public class Block {
2    Block north;
3    Block south;
4    Layer layer;
5    int width;
6    public Block(int width) {
7      this.width = width;
8      if (width > 2) {
9        north = new Block(width / 2);
10       south = new Block(width / 2);
11     }
12     layer = new Layer(width);
13   }
14   public int traverse(int input) {
15     int wire = layer.traverse(input);
16     if (width > 2) {
17       if (wire < width / 2) {
18         return north.traverse(wire);
19       } else {
20         return (width / 2) + south.traverse(wire - (width / 2));
21       }
22     } else {
23       return wire;
24     }
25   }
26 }
```

Figure 12.21 The BLOCK [w] network.

```
1  public class Periodic {
2    Block[] block;
3    public Periodic(int width) {
4      int logSize = 0;
5      int myWidth = width;
6      while (myWidth > 1) {
7        logSize++;
8        myWidth = myWidth / 2;
9      }
10     block = new Block[logSize];
11     for (int i = 0; i < logSize; i++) {
12       block[i] = new Block(width);
13     }
14   }
15   public int traverse(int input) {
16     int wire = input;
17     for (Block b : block) {
18       wire = b.traverse(wire);
19     }
20     return wire;
21   }
22 }
```

Figure 12.22 The Periodic network.

12.6 Diffracting Trees

Counting networks provide a high degree of pipelining, so throughput is largely independent of network depth. Latency, however, does depend on network depth. Of the counting networks we have seen, the most shallow has depth $\Theta(\log^2 w)$. Can we design a logarithmic-depth counting network? The good news is yes, such networks exist, but the bad news is that for all known constructions, the constant factors involved render these constructions impractical.

Here is an alternative approach. Consider a set of balancers with a single input wire and two output wires, with the top and bottom labeled 0 and 1, respectively. The TREE $[w]$ network (depicted in Fig. 12.23) is a binary tree structured as follows. Let w be a power of two, and define TREE $[2k]$ inductively. When k is equal to 1, TREE $[2k]$ consists of a single balancer with output wires y_0 and y_1. For $k > 1$, construct TREE $[2k]$ from two TREE $[k]$ trees and one additional balancer. Make the input wire x of the single balancer the root of the tree and connect each of its output wires to the input wire of a tree of width k. Redesignate output wires $y_0, y_1, \ldots, y_{k-1}$ of the TREE $[k]$ subtree extending from the "0" output wire as the even output wires $y_0, y_2, \ldots, y_{2k-2}$ of the final TREE $[2k]$ network and the wires $y_0, y_1, \ldots, y_{k-1}$ of the TREE $[k]$ subtree extending from the balancer's "1" output wire as the odd output wires $y_1, y_3, \ldots, y_{2k-1}$ of final TREE $[2k]$ network.

To understand why the TREE $[2k]$ network has the step property in a quiescent state, let us assume inductively that a quiescent TREE $[k]$ has the step property. The root balancer passes at most one token more to the TREE $[k]$ subtree on its "0" (top) wire than on its"1" (bottom) wire. The tokens exiting the top TREE $[k]$ subtree have a step property differing from that of the bottom subtree at exactly one wire j among their k output wires. The TREE $[2k]$ outputs are a perfect shuffle of the wires leaving the two subtrees, and it follows that the two step-shaped token sequences of width k form a new step of width $2k$ where the possible single

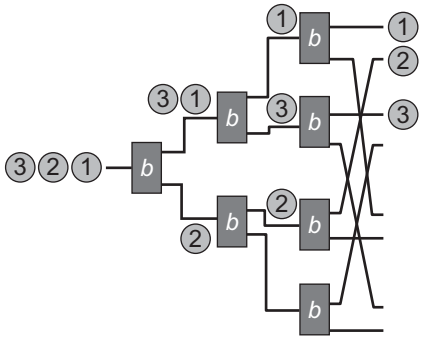

Figure 12.23 The TREE [8] class: a tree that counts. Notice how the network maintains the step property.

excess token appears at the higher of the two wires j, that is, the one from the top TREE $[k]$ tree.

The TREE $[w]$ network may be a counting network, but is it a *good* counting network? The good news is that it has shallow depth: while a BITONIC $[w]$ network has depth $\log^2 w$, the TREE $[w]$ network depth is just $\log w$. The bad news is contention: every token that enters the network passes through the same root balancer, causing that balancer to become a bottleneck. In general, the higher the balancer in the tree, the higher the contention.

We can reduce contention by exploiting a simple observation similar to one we made about the `EliminationBackoffStack` of Chapter 11:

> If an *even* number of tokens pass through a balancer, the outputs are evenly balanced on the top and bottom wires, but the balancer's state remains unchanged.

The basic idea behind *diffracting trees* is to place a `Prism` at each balancer, an out-of-band mechanism similar to the `EliminationArray` which allowed tokens (threads) accessing a stack to exchange items. The `Prism` allows tokens to pair off at random array locations and agree to diffract in different directions, that is, to exit on different wires without traversing the balancer's toggle bit or changing its state. A token traverses the balancer's toggle bit only if it is unable to pair off with another token within a reasonable period of time. If it did not manage to diffract, the token toggles the bit to determine which way to go. It follows that we can avoid excessive contention at balancers if the prism can pair off enough tokens without introducing too much contention.

A `Prism` is an array of `Exchanger<Integer>` objects, like the `EliminationArray`. An `Exchanger<T>` object permits two threads to exchange `T` values. If thread A calls the object's `exchange`() method with argument a, and B calls the same object's `exchange`() method with argument b, then A's call returns value b and vice versa. The first thread to arrive is blocked until the second arrives. The call includes a timeout argument allowing a thread to proceed if it is unable to exchange a value within a reasonable duration.

The `Prism` implementation appears in Fig. 12.24. Before thread A visits the balancer's toggle bit, it visits associated `Prism`. In the `Prism`, it picks an array entry at random, and calls that slot's `exchange`() method, providing its own thread ID as an exchange value. If it succeeds in exchanging ids with another thread, then the lower thread ID exits on wire 0, and the higher on wire 1.

Fig. 12.24 shows a `Prism` implementation. The constructor takes as an argument the capacity of the prism (the maximal number of distinct exchangers). The `Prism` class provides a single method, `visit`(), that chooses the random exchanger entry. The `visit`() call returns *true* if the caller should exit on the top wire, *false* if the bottom wire, and it throws a `TimeoutException` if the timeout expires without exchanging a value. The caller acquires its thread ID (Line 13), chooses a random entry in the array (Line 14), and tries to exchange its own ID with its partner's (Line 15). If it succeeds, it returns a Boolean value, and if it times out, it rethrows `TimeoutException`.

```
1  public class Prism {
2    private static final int duration = 100;
3    Exchanger<Integer>[] exchanger;
4    Random random;
5    public Prism(int capacity) {
6      exchanger = (Exchanger<Integer>[]) new Exchanger[capacity];
7      for (int i = 0; i < capacity; i++) {
8        exchanger[i] = new Exchanger<Integer>();
9      }
10     random = new Random();
11   }
12   public boolean visit() throws TimeoutException, InterruptedException {
13     int me = ThreadID.get();
14     int slot = random.nextInt(exchanger.length);
15     int other = exchanger[slot].exchange(me, duration, TimeUnit.MILLISECONDS);
16     return (me < other);
17   }
18 }
```

Figure 12.24 The `Prism` class.

```
1  public class DiffractingBalancer {
2    Prism prism;
3    Balancer toggle;
4    public DiffractingBalancer(int capacity) {
5      prism = new Prism(capacity);
6      toggle = new Balancer();
7    }
8    public int traverse() {
9      boolean direction = false;
10     try{
11       if (prism.visit())
12         return 0;
13       else
14         return 1;
15     } catch(TimeoutException ex) {
16       return toggle.traverse();
17     }
18   }
19 }
```

Figure 12.25 The `DiffractingBalancer` class: if the caller pairs up with a concurrent caller through the prism, it does not need to traverse the balancer.

A `DiffractingBalancer` (Fig. 12.25), like a regular `Balancer`, provides a `traverse()` method whose return value alternates between 0 and 1. This class has two fields: `prism` is a `Prism`, and `toggle` is a `Balancer`. When a thread calls `traverse()`, it tries to find a partner through the `prism`. If it succeeds, then the partners return with distinct values, without creating contention at the `toggle`

```
1  public class DiffractingTree {
2    DiffractingBalancer root;
3    DiffractingTree[] child;
4    int size;
5    public DiffractingTree(int mySize) {
6      size = mySize;
7      root = new DiffractingBalancer(size);
8      if (size > 2) {
9        child = new DiffractingTree[]{
10         new DiffractingTree(size/2),
11         new DiffractingTree(size/2)};
12     }
13   }
14   public int traverse() {
15     int half = root.traverse();
16     if (size > 2) {
17       return (2 * (child[half].traverse()) + half);
18     } else {
19       return half;
20     }
21   }
22 }
```

Figure 12.26 The `DiffractingTree` class: fields, constructor, and `traverse()` method.

(Line 11). Otherwise, if the thread is unable to find a partner, it traverses (Line 16) the `toggle` (implemented as a balancer).

The `DiffractingTree` class (Fig. 12.26) has two fields. The `child` array is a two-element array of child trees. The `root` field is a `DiffractingBalancer` that alternates between forwarding calls to the left or right subtree. Each `DiffractingBalancer` has a capacity, which is actually the capacity of its internal prism. Initially this capacity is the size of the tree, and the capacity shrinks by half at each level.

As with the `EliminationBackoffStack`, `DiffractingTree` performance depends on two parameters: prism capacities and timeouts. If the prisms are too big, threads miss one another, causing excessive contention at the balancer. If the arrays are too small, then too many threads concurrently access each exchanger in a prism, resulting in excessive contention at the exchangers. If prism timeouts are too short, threads miss one another, and if they are too long, threads may be delayed unnecessarily. There are no hard-and-fast rules for choosing these values, since the optimal values depend on the load and the characteristics of the underlying multiprocessor architecture.

Nevertheless, experimental evidence suggests that it is sometimes possible to choose these values to outperform both the `CombiningTree` and `CountingNetwork` classes. Here are some heuristics that work well in practice. Because balancers higher in the tree have more contention, we use larger prisms near the top of the tree, and add the ability to dynamically shrink and grow the

random range chosen. The best timeout interval choice depends on the load: if only a few threads are accessing the tree, then time spent waiting is mostly wasted, while if there are many threads, then time spent waiting pays off. Adaptive schemes are promising: lengthen the timeout while threads succeed in pairing off, and shorten it otherwise.

12.7 Parallel Sorting

Sorting is one of the most important computational tasks, dating back to Hollerith's Nineteenth-Century sorting machine, through the first electronic computer systems in the 1940s, and culminating today, when a high fraction of programs use sorting in some form or another. As most Computer Science undergraduates learn early on, the choice of sorting algorithm depends crucially on the number of items being sorted, the numerical properties of their keys, and whether the items reside in memory or in an external storage device. Parallel sorting algorithms can be classified in the same way.

We present two classes of sorting algorithms: *sorting networks*, which typically work well for small in-memory data sets, and *sample sorting algorithms*, which work well for large data sets in external memory. In our presentation, we sacrifice performance for simplicity. More complex techniques are cited in the chapter notes.

12.8 Sorting Networks

In much the same way that a counting network is a network of *balancers*, a sorting network is a network of *comparators*.[3] A comparator is a computing element with two input wires and two output wires, called the *top* and *bottom* wires. It receives two numbers on its input wires, and forwards the larger to its top wire and the smaller to its bottom wire. A comparator, unlike a balancer, is *synchronous*: it outputs values only when both inputs have arrived (see Fig. 12.27).

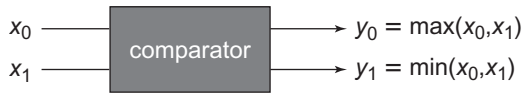

Figure 12.27 A comparator.

3 Historically sorting networks predate counting networks by several decades.

A *comparison network*, like a balancing network, is an acyclic network of comparators. An input value is placed on each of its w input lines. These values pass through each layer of comparators synchronously, finally leaving together on the network output wires.

A comparison network with input values x_i and output values y_i, $i \in \{0 \ldots 1\}$, each on wire i, is a valid *sorting network* if its output values are the input values sorted in descending order, that is, $y_{i-1} \geqslant y_i$.

The following classic theorem simplifies the process of proving that a given network sorts.

***Theorem* 12.8.1 (0-1-principle).** If a sorting network sorts every input sequence of 0s and 1s, then it sorts any sequence of input values.

12.8.1 Designing a Sorting Network

There is no need to design sorting networks, because we can recycle counting network layouts. A balancing network and a comparison network are *isomorphic* if one can be constructed from the other by replacing balancers with comparators, or vice versa.

***Theorem* 12.8.2.** If a balancing network counts, then its isomorphic comparison network sorts.

Proof: We construct a mapping from comparison network transitions to isomorphic balancing network transitions.

By Theorem 12.8.1, a comparison network which sorts all sequences of 0s and 1s is a sorting network. Take any arbitrary sequence of 0s and 1s as inputs to the comparison network, and for the balancing network place a token on each 1 input wire and no token on each 0 input wire. If we run both networks in lock-step, the balancing network simulates the comparison network.

The proof is by induction on the depth of the network. For level 0 the claim holds by construction. Assuming it holds for wires of a given level k, let us prove it holds for level $k+1$. On every comparator where two 1s meet in the comparison network, two tokens meet in the balancing network, so one 1 leaves on each wire in the comparison network on level $k+1$, and one token leaves on each wire in the balancing network on level $k+1$. On every comparator where two 0s meet in the comparison network, no tokens meet in the balancing network, so a 0 leaves on each level $k+1$ wire in the comparison network, and no tokens leave in the balancing network. On every comparator where a 0 and 1 meet in the comparison network, the 1 leaves on the north (upper) wire and the 1 on the south (lower) wire on level $k+1$, while in the balancing network the token leaves on the north wire, and no token leaves on the south wire.

If the balancing network is a counting network, that is, it has the step property on its output level wires, then the comparison network must have sorted the input sequence of 0s and 1s. □

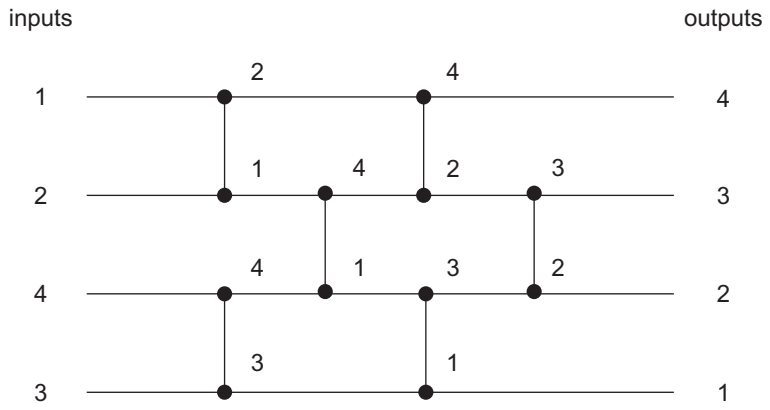

Figure 12.28 The `OddEven` sorting network.

The converse is false: not all sorting networks are counting networks. We leave it as an exercise to verify that the `OddEven` network in Fig. 12.28 is a sorting network but not a counting network.

***Corollary* 12.8.1.** Comparison networks isomorphic to BITONIC [] and PERIODIC [] networks are sorting networks.

Sorting a set of size w by comparisons requires $\Omega(w \log w)$ comparisons. A sorting network with w input wires has at most $O(w)$ comparators in each level, so its depth can be no smaller than $\Omega(\log w)$.

***Corollary* 12.8.2.** The depth of any counting network is at least $\Omega(\log w)$.

A Bitonic Sorting Algorithm

We can represent any width-w sorting network, such as BITONIC [w], as a collection of d layers of $w/2$ balancers each. We can represent a sorting network layout as a table, where each entry is a pair that describes which two wires meet at that balancer at that layer. (E.g., in the BITONIC [4] network of Fig. 12.11, wires 0 and 1 meet at the first balancer in the first layer, and wires 0 and 3 meet at the first balancer of the second layer.) Let us assume, for simplicity, that we are given an unbounded table `bitonicTable`[i][d][j], where each array entry contains the index of the associated north (0) or south (1) input wire to balancer i at depth d.

An *in-place* array-based sorting algorithm takes as input an array of items to be sorted (here we assume these items have unique integer keys) and returns the same array with the items sorted by key. Here is how we implement `BitonicSort`, an in-place array-based sorting algorithm based on a Bitonic

sorting network. Let us assume that we wish to sort an array of $2 \cdot p \cdot s$ elements, where p is the number of threads (and typically also the maximal number of available processors on which the threads run) and $p \cdot s$ is a power of 2. The network has $p \cdot s$ comparators at every layer.

Each of the p threads emulates the work of s comparators. Unlike counting networks, which act like uncoordinated raves, sorting networks are synchronous: all inputs to a comparator must arrive before it can compute the outputs. The algorithm proceeds in rounds. In each round, a thread performs s comparisons in a layer of the network, switching the array entries of items if necessary, so that they are properly ordered. In each network layer, the comparators join different wires, so no two threads attempt to exchange the items of the same entry, avoiding the need to synchronize operations at any given layer.

To ensure that the comparisons of a given round (layer) are complete before proceeding to the next one, we use a synchronization construct called a `Barrier` (studied in more detail in Chapter 17). A barrier for p threads provides an `await()` method, whose call does not return until all p threads have called `await()`. The `BitonicSort` implementation appears in Fig. 12.29. Each thread proceeds through the layers of the network round by round. In each round, it awaits the arrival of the other threads (Line 12), ensuring that the `items` array contains the prior round's results. It then emulates the behavior of s balancers at that layer by comparing the items at the array positions corresponding to the

```
1  public class BitonicSort {
2    static final int[][][] bitonicTable = ...;
3    static final int width = ...; // counting network width
4    static final int depth = ...; // counting network depth
5    static final int p = ...;  // number of threads
6    static final int s = ...;  // a power of 2
7    Barrier barrier;
8    ...
9      public <T> void sort(Item<T>[] items) {
10       int i = ThreadID.get();
11       for (int d = 0; d < depth; d++) {
12         barrier.await();
13         for (int j = 0; j < s; j++) {
14           int north = bitonicTable[(i*s)+j][d][0];
15           int south = bitonicTable[(i*s)+j][d][1];
16           if (items[north].key < items[south].key) {
17             Item<T> temp = items[north];
18             items[north] = items[south];
19             items[south] = temp;
20           }
21         }
22       }
23     }
```

Figure 12.29 The `BitonicSort` class.

comparator's wires, and exchanging them if their keys are out of order (Lines 14 through 19).

The `BitonicSort` takes $O(s \log^2 p)$ time for p threads running on p processors, which, if s is constant, is $O(\log^2 p)$ time.

12.9 Sample Sorting

The `BitonicSort` is appropriate for small data sets that reside in memory. For larger data sets (where n, the number of items, is much larger than p, the number of threads), especially ones that reside on out-of-memory storage devices, we need a different approach. Because accessing a data item is expensive, we must maintain as much locality-of-reference as possible, so having a single thread sort items sequentially is cost-effective. A parallel sort like `BitonicSort`, where an item is accessed by multiple threads, is simply too expensive.

We attempt to minimize the number of threads that access a given item through randomization. This use of randomness differs from that in the `DiffractingTree`, where it was used to distribute memory accesses. Here we use randomness to guess the distribution of items in the data set to be sorted.

Since the data set to be sorted is large, we split it into buckets, throwing into each bucket the items that have keys within a given range. Each thread then sorts the items in one of the buckets using a sequential sorting algorithm, and the result is a sorted set (when viewed in the appropriate bucket order). This algorithm is a generalization of the well-known *quicksort* algorithm, but instead of having a single *splitter* key to divide the items into two subsets, we have $p - 1$ splitter keys that split the input set into p subsets.

The algorithm for n items and p threads involves three phases:

1. Threads choose $p - 1$ splitter keys to partition the data set into p buckets. The splitters are published so all threads can read them.
2. Each thread sequentially processes n/p items, moving each item to its bucket, where the appropriate bucket is determined by performing a binary search with the item's key among the splitter keys.
3. Each thread sequentially sorts the items in its bucket.

Barriers between the phases ensure that all threads have completed one phase before the next starts.

Before we consider Phase one, we look at the second and third phases.

The second phase's time complexity is $(n/p) \log p$, consisting of reading each item from memory, disk, or tape, followed by a binary search among p splitters cached locally, and finally adding the item into the appropriate bucket. The buckets into which the items are moved could be in memory, on disk, or on tape, so the dominating cost is that of the n/p accesses to the stored data items.

Let b be the number of items in a bucket. The time complexity of the third phase for a given thread is $O(b \log b)$, to sort the items using a sequential version of, say, *quicksort*.[4] This part has the highest cost because it consists of read–write phases that access relatively slow memory, such as disk or tape.

The time complexity of the algorithm is dominated by the thread with the most items in its bucket in the third phase. It is therefore important to choose the splitters to be as evenly distributed as possible, so each bucket receives approximately $n - p$ items in the second phase.

The key to choosing good splitters is to have each thread pick a set of *sample* splitters that represent its own $n - p$ size data set, and choose the final $p - 1$ splitters from among all the sample splitter sets of all threads. Each thread selects uniformly at random s keys from its data set of size $n - p$. (In practice, it suffices to choose s to be 32 or 64 keys.) Each thread then participates in running the parallel `BitonicSort` (Fig. 12.29) on the $s \cdot p$ sample keys selected by the p threads. Finally, each thread reads the $p - 1$ splitter keys in positions $s, 2s, \ldots, (p-1)s$ in the sorted set of splitters, and uses these as the splitters in the second phase. This choice of s samples, and the later choice of the final splitters from the sorted set of all samples, reduces the effects of an uneven key distribution among the $n - p$ size data sets accessed by the threads.

For example, a sample sort algorithm could choose to have each thread pick $p - 1$ splitters for its second phase from within its own n/p size data set, without ever communicating with other threads. The problem with this approach is that if the distribution of the data is uneven, the size of the buckets may differ greatly, and performance would suffer. For example, if the number of items in the largest bucket is doubled, so is the worst-case time complexity of sorting algorithm.

The first phase's complexity is s (a constant) to perform the random sampling, and $O(\log^2 p)$ for the parallel Bitonic sort. The overall time complexity of sample sort with a good splitter set (where every bucket gets $O(n/p)$ of the items) is

$$O(\log^2 p) + O((n/p) \log p) + O((n/p) \log(n/p))$$

which overall is $O((n/p) \log(n/p))$.

12.10 Distributed Coordination

This chapter covered several distributed coordination patterns. Some, such as combining trees, sorting networks, and sample sorting, have high parallelism and low overheads. All these algorithms contain synchronization bottlenecks, that is, points in the computation where threads must wait to rendezvous with others. In the combining trees, threads must synchronize to combine, and in sorting, when threads wait at barriers.

---

4 If the item's key size is known and fixed, one could use algorithms like *Radixsort*.

In other schemes, such as counting networks and diffracting trees, threads never wait for one another. (Although we implement balancers using **synchronized** methods, they could be implemented in a lock-free manner using `compareAndSet()`.) Here, the distributed structures pass information from one thread to another, and while a rendezvous could prove advantageous (as in the `Prism` array), it is not necessary.

Randomization, which is useful in many places, helps to distribute work evenly. For diffracting trees, randomization distributes work over multiple memory locations, reducing the chance that too many threads simultaneously access the same location. For sample sort, randomization helps distribute work evenly among buckets, which threads later sort in parallel.

Finally, we saw that pipelining can ensure that some data structures can have high throughput, even though they have high latency.

Although we focus on shared-memory multiprocessors, it is worth mentioning that the distributed algorithms and structures considered in this chapter also work in message-passing architectures. The message-passing model might be implemented directly in hardware, as in a network of processors, or it could be provided on top of a shared-memory architecture through a software layer such as MPI.

In shared-memory architectures, switches (such as combining tree nodes or balancers) are naturally implemented as shared-memory counters. In message-passing architectures, switches are naturally implemented as processor-local data structures, where wires that link one processor to another also link one switch to another. When a processor receives a message, it atomically updates its local data structure and forwards messages to the processors managing other switches.

12.11 Chapter Notes

The idea behind combining trees is due to Allan Gottlieb, Ralph Grishman, Clyde Kruskal, Kevin McAuliffe, Larry Rudolph, and Marc Snir [47]. The software `CombiningTree` presented here is a adapted from an algorithm by PenChung Yew, Nian-Feng Tzeng, and Duncan Lawrie [151] with modifications by Maurice Herlihy, Beng-Hong Lim, and Nir Shavit [65], all based on an original proposal by James Goodman, Mary Vernon, and Philip Woest [45].

Counting networks were invented by Jim Aspnes, Maurice Herlihy, and Nir Shavit [16]. Counting networks are related to *sorting networks*, including the ground breaking Bitonic network of Kenneth Batcher [18], and the periodic network of Martin Dowd, Yehoshua Perl, Larry Rudolph, and Mike Saks [35]. Miklós Ajtai, János Komlós, and Endre Szemerédi discovered the AKS sorting network, an $O(\log w)$ depth sorting network [8]. (This asymptotic expression hides large constants which make networks based on AKS impractical.)

Mike Klugerman and Greg Plaxton [84, 85] were the first to provide an AKS-based counting network construction with $O(\log w)$ depth. The 0-1 principle for sorting networks is by Donald Knuth [86]. A similar set of rules for balancing networks is provided by Costas Busch and Marios Mavronicolas [25]. Diffracting trees were invented by Nir Shavit and Asaph Zemach [143].

Sample sorting was suggested by John Reif and Leslie Valiant [132] and by Huang and Chow [73]. The sequential Quicksort algorithm to which all sample sorting algorithms relate is due to Tony Hoare [70]. There are numerous parallel radix sort algorithms in the literature such as the one by Daniel Jiménez-González, Joseph Larriba-Pey, and Juan Navarro [82] or the one by Shin-Jae Lee and Minsoo Jeon and Dongseung Kim and Andrew Sohn [101].

Monty Python and the Holy Grail was written by Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones, and Michael Palin and co-directed by Terry Gilliam and Terry Jones [27].

12.12 Exercises

Exercise 134. Prove Lemma 12.5.1.

Exercise 135. Implement a *trinary* `CombiningTree`, that is, one that allows up to three threads coming from three subtrees to combine at a given node. Can you estimate the advantages and disadvantages of such a tree when compared to a *binary* combining tree?

Exercise 136. Implement a `CombiningTree` using `Exchanger` objects to perform the coordination among threads ascending and descending the tree. What are the possible disadvantages of your construction when compared to the `CombiningTree` class presented in Section 12.3?

Exercise 137. Implement the cyclic array based shared pool described in Section 12.2 using two simple counters and a `ReentrantLock` per array entry.

Exercise 138. Provide an efficient lock-free implementation of a `Balancer`.

Exercise 139. (Hard) Provide an efficient wait-free implementation of a `Balancer` (i.e. not by using the universal construction).

Exercise 140. Prove that the TREE $[2k]$ balancing network constructed in Section 12.6 is a counting network, that is, that in any quiescent state, the sequences of tokens on its output wires have the step property.

Exercise 141. Let $\mathcal{B}$ be a width-w balancing network of depth d in a quiescent state s. Let $n = 2^d$. Prove that if n tokens enter the network on the same wire, pass through the network, and exit, then $\mathcal{B}$ will have the same state after the tokens exit as it did before they entered.

In the following exercises, a *k-smooth sequence* is a sequence $y_0, \ldots, y_{w-1}$ that satisfies

$$\text{if} \quad i < j \quad \text{then} \quad |y_i - y_j| \leqslant k.$$

Exercise 142. Let X and Y be k-smooth sequences of length w. A *matching* layer of balancers for X and Y is one where each element of X is joined by a balancer to an element of Y in a one-to-one correspondence.

Prove that if X and Y are each k-smooth, and Z is the result of matching X and Y, then Z is $(k + 1)$-smooth.

Exercise 143. Consider a BLOCK$[k]$ network in which each balancer has been initialized to an arbitrary state (either *up* or *down*). Show that no matter what the input distribution is, the output distribution is $(\log k)$-smooth.

Hint: you may use the claim in Exercise 142.

Exercise 144. A *smoothing network* is a balancing network that ensures that in any quiescent state, the output sequence is 1-smooth.

Counting networks are smoothing networks, but not vice versa.

A Boolean sorting network is one in which all inputs are guaranteed to be Boolean. Define a *pseudo-sorting balancing network* to be a balancing network with a layout isomorphic to a Boolean sorting network.

Let $\mathcal{N}$ be the balancing network constructed by taking a smoothing network $\mathcal{S}$ of width w, a pseudo-sorting balancing network $\mathcal{P}$ also of width w, and joining the i^{th} output wire of $\mathcal{S}$ to the i^{th} input wire of $\mathcal{P}$.

Show that $\mathcal{N}$ is a counting network.

Exercise 145. A *3-balancer* is a balancer with three input lines and three output lines. Like its 2-line relative, its output sequences have the step property in any quiescent state. Construct a depth-3 counting network with 6 input and output lines from 2-balancers and 3-balancers. Explain why it works.

Exercise 146. Suggest ways to modify the `BitonicSort` class so that it will sort an input array of width w where w is not a power of 2.

Exercise 147. Consider the following w-thread counting algorithm. Each thread first uses a bitonic counting network of width w to take a counter value v. It then goes through a *waiting filter*, in which each thread waits for threads with lesser values to catch up.

The waiting filter is an array `filter[]` of w Boolean values. Define the phase function

$$\phi(v) = \lfloor (v/w) \rfloor \bmod 2.$$

A thread that exits with value v spins on `filter`$[(v-1) \bmod n]$ until that value is set to $\phi(v-1)$. The thread responds by setting `filter`$[v \bmod w]$ to $\phi(v)$, and then returns v.

1. Explain why this counter implementation is linearizable.
2. An exercise here shows that any linearizable counting network has depth at least w. Explain why the `filter[]` construction does not contradict this claim.
3. On a bus-based multiprocessor, would this `filter[]` construction have better throughput than a single variable protected by a spin lock? Explain.

Exercise 148. If a sequence $X = x_0, \ldots x_{w-1}$ is k-smooth, then the result of passing X through a balancing network is k-smooth.

Exercise 149. Prove that the `Bitonic`$[w]$ network has depth $(\log w)(1 + \log w)/2$ and uses $(w \log w)(1 + \log w)/4$ balancers.

Exercise 150. (Hard) Provide an implementation of a `DiffractingBalancer` that is lock-free.

Exercise 151. Add an adaptive timeout mechanism to the `Prism` of the `DiffractingBalancer`.

Exercise 152. Show that the `OddEven` network in Fig. 12.28 is a sorting network but not a counting network.

Exercise 153. Can counting networks do anything besides increments? Consider a new kind of token, called an *antitoken*, which we use for decrements. Recall that when a token visits a balancer, it executes a `getAndComplement()`: it atomically reads the toggle value and complements it, and then departs on the output wire indicated by the old toggle value. Instead, an antitoken complements the toggle value, and then departs on the output wire indicated by the new toggle value. Informally, an antitoken "cancels" the effect of the most recent token on the balancer's toggle state, and vice versa.

Instead of simply balancing the number of tokens that emerge on each wire, we assign a *weight* of +1 to each token and −1 to each antitoken. We generalize the step property to require that the sums of the weights of the tokens and antitokens that emerge on each wire have the step property. We call this property the *weighted step property*.

```
1   public synchronized int antiTraverse() {
2     try {
3       if (toggle) {
4         return 1;
5       } else {
6         return 0;
7       }
8     } finally {
9       toggle = !toggle;
10    }
11  }
```

Figure 12.30 The `antiTraverse()` method.

Fig. 12.30 shows how to implement an `antiTraverse()` method that moves an antitoken though a balancer. Adding an `antiTraverse()` method to the other networks is left as an exercise.

Let $\mathcal{B}$ be a width-w balancing network of depth d in a quiescent state s. Let $n = 2^d$. Show that if n tokens enter the network on the same wire, pass through the network, and exit, then $\mathcal{B}$ will have the same state after the tokens exit as it did before they entered.

Exercise 154. Let $\mathcal{B}$ be a balancing network in a quiescent state s, and suppose a token enters on wire i and passes through the network, leaving the network in state s'. Show that if an antitoken now enters on wire i and passes through the network, then the network goes back to state s.

Exercise 155. Show that if balancing network $\mathcal{B}$ is a counting network for tokens alone, then it is also a balancing network for tokens and antitokens.

Exercise 156. A *switching network* is a directed graph, where edges are called *wires* and node are called *switches*. Each thread shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire. Here, for simplicity, switches have two input and output wires. Note that switching networks are more powerful than balancing networks, since switches can have arbitrary state (instead of a single bit) and tokens also have state.

An *adding network* is a switching network that allows threads to add (or subtract) arbitrary values.

We say that a token is *in front of* a switch if it is on one of the switch's input wires. Start with the network in a quiescent state q_0, where the next token to run will take value 0. Imagine we have one token t of weight a and n–1 tokens $t_1, \ldots, t_{n-1}$ all of weight b, where $b > a$, each on a distinct input wire. Denote by $\mathcal{S}$ the set of switches that t traverses if it traverses the network by starting in q_0.

Prove that if we run the $t_1, \ldots, t_{n-1}$ one at a time though the network, we can halt each t_i in front of a switch of S.

At the end of this construction, $n - 1$ tokens are in front of switches of S. Since switches have two input wires, it follows that t's path through the network encompasses at least $n - 1$ switches, so any adding network must have depth at least $n - 1$, where n is the maximum number of concurrent tokens. This bound is discouraging because it implies that the size of the network depends on the number of threads (also true for `CombiningTree`s, but not counting networks), and that the network has inherently high latency.

Exercise 157. Extend the proof of Exercise 156 to show that a *linearizable* counting network has depth at least n.