Multiprocessor Synchronization and Concurrent Data Structures

Maurice Herlihy Nir Shavit

September 4, 2005

Chapter 3

Mutual Exclusion

This chapter covers a number of “classical” mutual exclusion algorithms that work by reading and writing shared memory. Although these algorithms are not used in practice, we study them because they provide an ideal introduction to the kinds of correctness issues that arise in every area of synchronization. These algorithms, simple as they are, display subtle properties that should be understood before we are ready to approach the design of truly practical techniques.

3.1 Time

Reasoning about concurrent computation is mostly reasoning about time. Sometimes we want things to happen at the same time, and sometimes we want them to happen at different times. We will need to reason about complicated conditions involving how multiple time intervals can overlap, or, sometimes, how they can’t. We need a simple but unambiguous language to talk about events and durations in time. Everyday English is too ambiguous and imprecise. Instead, we will introduce a simple vocabulary and notation to describe how concurrent threads behave in time.

In 1689, Isaac Newton stated “absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” We endorse his notion of time, if not his prose style. Threads share a common time (though not necessarily a common clock). A thread is a *state machine*, and its state transitions are called *events*. Events are

[0] *This chapter is part of the Manuscript* MULTIPROCESSOR SYNCHRONIZATION AND CONCURRENT DATA STRUCTURES *by Maurice Herlihy and Nir Shavit copyright © 2005, all rights reserved.*

```
class Counter {
  private int value;
  public Counter(int c) {        // constructor
    value = c;
  }
  // increment and return prior value
  public int getAndIncrement() {
    int temp = value;            // start of danger zone
    value    = temp + 1;         // end of danger zone
    return temp;
  }
}
```

Figure 3.1: The `Counter` Class

instantaneous: they occur at a single instant of time. Events are never simultaneous: distinct events occur at distinct times. A thread A may produce a sequence of events $a_0, a_1, \ldots$ Threads typically contain loops, so a single program statement can produce many events. We denote the j-th occurrence of an event a_i by a_i^j. One event a *precedes* another event b, written $a \to b$, if a occurs at an earlier time. The *precedence* relation "$\to$" is a total order on events.

Let a_0 and a_1 be events such that $a_0 \to a_1$. An *interval* (a_0, a_1) is the duration between a_0 and a_1. Interval $I_A = (a_0, a_1)$ *precedes* $I_B = (b_0, b_1)$, written $I_A \to I_B$, if $a_1 \to b_0$ (that is, if the final event of I_A precedes the starting event of I_B). More succinctly, the $\to$ relation is a partial order on intervals. Intervals that are unrelated by $\to$ are said to be *concurrent*. By analogy with events, we denote the j-th execution of interval I_A by I_A^j.

3.2 Critical Sections

In an earlier chapter, we discussed the `Counter` class implementation shown in Figure 3.1. We observed that this implementation is correct in a single-thread system, but misbehaves when used by two or more threads. The problem occurs if two threads both read the `value` field at the line marked "start of danger zone", and then both update that field at the line marked "end of danger zone".

We can avoid this problem if we transform these two lines into a *critical*

```
public interface Lock {
  public void lock();      // before entering critical section
  public void unlock();    // before leaving critical section
}
```

Figure 3.2: The `Lock` Interface

section: a block of code that can be executed by only one thread at a time. We call this property *mutual exclusion*. The standard way to approach mutual exclusion is through a `Lock` object satisfying the interface shown in Figure 3.2.

We call such a mechanism a *lock*. For brevity, we say a thread *acquires* a lock when it executes a `lock()` method call, and *releases* the lock when it executes an `unlock()` method call. Figure 3.3 shows how to use a `Lock` field to add mutual exclusion to a shared counter implementation. Threads using the `lock()` and `unlock()` methods must follow a specific format. A thread is *well-formed* if

1. each critical section is associated with a unique `Lock` object,
2. the thread calls that object's `lock()` method when it is trying to enter the critical section, and
3. the thread calls the `unlock()` method when it leaves the critical section.

In Java, these methods should be used in the following structured way.

```
l.lock();
try {
  ...                  // body
  } finally {
    l.unlock();
  }
```

This idiom ensures that (1) the lock has been acquired before entering the try block, and (2) the lock will be released no matter what exception is thrown by any statement in the body.

We now formalize the properties a good `Lock` algorithm should satisfy. Let CS_A^j be the interval during which A executes the critical section for the j-th time. Assume, for simplicity, that each thread acquires and releases the lock infinitely often, with other work taking place in the meantime.

```
public class SafeCounter {
  private long value;
  private Lock lock;              // to protect critical section

  public long getAndIncrement() {
    lock.lock();                  // enter critical section
    long temp = value;            // in critical section
    value = temp + 1;             // in critical section
    lock.unlock();                // leave critical section
    return temp;
  }
}
```

Figure 3.3: Using a Lock Object

Mutual Exclusion Critical sections of different threads do not overlap. For threads A and B, and integers j and k, either $CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$.

No Deadlock If some thread attempts to acquire the lock, then some thread will acquire the lock. If thread A calls `lock()` but never acquires the lock, then other threads be completing an infinite number of critical sections.

No Lockout Every thread that attempts to acquire the lock eventually succeeds. Every call to `lock()` eventually returns. This property is sometimes called *no starvation*.

Note that the no-lockout property implies the no-deadlock property.

The mutual exclusion property is clearly essential. Without this property, we cannot guarantee that a computation's results are correct. In the terminology of Chapter 1, mutual exclusion is a safety property. The no-deadlock property is important. It implies that the system never "freezes." Individual threads may be stuck forever (called *starvation*), but some thread makes progress. In the terminology of Chapter 1, no-deadlock is a liveness property. Note that a program can still deadlock even if each of the locks it uses satisfies the no-deadlock property. For example, consider threads A and B that share locks ℓ_0 and ℓ_1. First, A acquires ℓ_0 and B acquires ℓ_1. Next, A tries to acquire ℓ_1 and B tries to acquire ℓ_0. The threads will deadlock because each one waits for the other to release its lock.

The lockout-free property, while clearly desirable, is the least compelling of the three. Later on, we will see "practical" mutual exclusion algorithms that fail to be lockout-free. These algorithms are typically deployed in circumstances where starvation is a theoretical possibility, but is unlikely to occur in practice. Nevertheless, the ability to reason about starvation is essential for understanding whether it is a realistic threat.

The lockout-free property is also weak in the sense that makes no guarantees of how long a thread will wait before it enters the critical section. Later on, we will look at algorithms that place bounds on how long a thread can wait.

3.3 Two-Thread Solutions

We begin with two inadequate but interesting `Lock` algorithms.

3.3.1 The `LockOne` Class

Figure 3.4 shows the `LockOne` algorithm. Our two-thread lock algorithms follow the following conventions: the threads have indexes 0 and 1, the calling thread has index i, and the other $j = 1 - i$. Each thread has a *thread-local* variable, called `myIndex`, that returns its index using a `get()` method.

We use $\text{write}_A(x = v)$ to denote the event in which A assigns value v to field x, and $\text{read}_A(v == x)$ to denote the event in which A reads v from field x. Sometimes we omit v when the value is unimportant. For example, in Figure 3.4 the event $\text{write}_A(\texttt{flag}[i] = true)$ is caused by the third line of the `lock()` method.

Lemma 3.3.1 *The `LockOne` algorithm satisfies mutual exclusion.*

Proof: Suppose not. Then there exist integers j and k such that $CS_A^j \not\rightarrow CS_B^k$ and $CS_B^k \not\rightarrow CS_A^j$. Consider each thread's last execution of the `lock()` method before entering its k-th (j-th) critical section.

Inspecting the code, we see that

$$\text{write}_A(\texttt{flag}[A] = true) \rightarrow \text{read}_A(\texttt{flag}[B] == false) \rightarrow CS_A \tag{3.1}$$

$$\text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{read}_B(\texttt{flag}[A] == false) \rightarrow CS_B \tag{3.2}$$

$$\text{read}_A(\texttt{flag}[B] == false) \rightarrow \text{write}_B(\texttt{flag}[B] = true) \tag{3.3}$$

Note that once $\texttt{flag}[B]$ is set to *true* it remains *true*. It follows that Equation 3.3 holds, since otherwise thread A could not have read $\texttt{flag}[B]$

```
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  // thread-local index, 0 or 1
  private static ThreadLocal<Integer> myIndex;
  public void lock() {
    int i = myIndex.get();
    int j = i-1;
    flag[i] = true;
    while (flag[j]) {}              // wait
  }
  public void unlock() {
    int i = myIndex.get();
    flag[i] = false;
  }
}
```

Figure 3.4: The `LockOne` Algorithm

as *false*. Equation 3.4 follows from Equations 3.1, 3.2, and 3.3, and because of the transitivity of the precedence order.

$$\text{write}_A(\texttt{flag}[A] = true) \rightarrow \text{read}_A(\texttt{flag}[B] == false) \rightarrow \\ \text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{read}_B(\texttt{flag}[A] == false) \tag{3.4}$$

It follows that $\text{write}_A(\texttt{flag}[A] = true) \rightarrow \text{read}_B(\texttt{flag}[A] == false)$ without an intervening write to the `flag` array, a contradiction. ■

The `LockOne` algorithm is inadequate because it deadlocks if thread executions are interleaved. If $\text{write}_A(\texttt{flag}[A] = true)$ and $\text{write}_B(\texttt{flag}[B] = true)$ events occur before $\text{read}_A(\texttt{flag}[B])$ and $\text{read}_B(\texttt{flag}[A])$ events, then both threads wait forever. Nevertheless, the `LockOne` class has an interesting property: if one thread runs before the other, no deadlock occurs, and all is well.

3.3.2 The `LockTwo` Class

Figure 3.5 shows an alternative lock algorithm, the `LockTwo` class.

Lemma 3.3.2 *The `LockTwo` algorithm satisfies mutual exclusion.*

```
class LockTwo
    implements Lock {
  private int victim;
  // thread-local index, 0 or 1
  private static ThreadLocal<Integer> myIndex;
  public void lock() {
    int i = myIndex.get();
    victim = i;                  // let the other go first
    while (victim == i) {}       // wait
  }
  public void unlock() {}
}
```

Figure 3.5: The `LockTwo` Algorithm

Proof: Suppose not. Then there exist integers j and k such that $CS_A^j \not\rightarrow CS_B^k$ and $CS_B^k \not\rightarrow CS_A^j$. Consider as before each thread's last execution of the `lock()` method before entering its k-th (j-th) critical section.

Inspecting the code, we see that

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{victim} == B) \rightarrow CS_A \qquad (3.5)$$
$$\text{write}_B(\texttt{victim} = B) \rightarrow \text{read}_B(\texttt{victim} == A) \rightarrow CS_B \qquad (3.6)$$

Thread B must assign B to the `victim` field between events $\text{write}_A(\texttt{victim} = A)$ and $\text{read}_A(\texttt{victim} = B)$ (see Equation 3.5). Since this assignment is the last, we have

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{write}_B(\texttt{victim} = B) \rightarrow \text{read}_A(\texttt{victim} == B) \qquad (3.7)$$

Once the `victim` field is set to B, it does not change, so any subsequent read will return B, contradicting Equation 3.6. ■

The `LockTwo` class is inadequate because it deadlocks if one thread runs completely before the other. Nevertheless, the `LockTwo` class satisfies an interesting property: if the threads run concurrently, the `lock()` method succeeds. The `LockOne` and `LockTwo` classes complement one another: each succeeds under conditions that cause the other to deadlock.

3.3.3 The Peterson Lock

We now combine the `LockOne` and `LockTwo` algorithms to construct a lockout-free `Lock` algorithm, shown in Figure 3.6. This algorithm is arguably the

```
class Peterson implements Lock {
  // thread-local index, 0 or 1
  private static ThreadLocal<Integer> myIndex;
  private boolean[] flag = new boolean[2];
  private int victim;
  public void lock() {
    int i = myIndex.get();
    int j = 1-i;
    flag[i] = true;                // I'm interested
    victim  = i;                   // you go first
    while (flag[j] && victim == i) {}; // wait
  }
  public void unlock() {
    int i = myIndex.get();
    flag[i] = false;               // I'm not interested
  }
}
```

Figure 3.6: The `Peterson` Lock Algorithm.

most succinct and elegant two-thread mutual exclusion algorithm. It is known as "Peterson's Algorithm", after its inventor.

We now sketch a correctness proof.

Lemma 3.3.3 *The `Peterson` lock algorithm satisfies mutual exclusion.*

Proof: Suppose not. As before, consider the last executions of the `lock()` method by threads A and B. Inspecting the code, we see that

$$\text{write}_A(\texttt{flag}[A] = true) \rightarrow \text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B]) \rightarrow \text{read}_A(\texttt{victim}) \rightarrow CS_A \tag{3.8}$$

$$\text{write}_B(\texttt{flag}[B] = true) \rightarrow \text{write}_B(\texttt{victim} = B) \rightarrow \text{read}_B(\texttt{flag}[A]) \rightarrow \text{read}_B(\texttt{victim}) \rightarrow CS_B \tag{3.9}$$

Assume, without loss of generality, that A was the last thread to write to the `victim` field.

$$\text{write}_B(\texttt{victim} = B) \rightarrow \text{write}_A(\texttt{victim} = A) \tag{3.10}$$

Equation 3.10 implies that A read `victim` to be A in Equation 3.8. Since A nevertheless entered its critical section, it must have read `flag[B]` to be *false*, so we have:

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false}) \tag{3.11}$$

Equations 3.9, 3.10, and 3.11, and transitivity of $\rightarrow$ imply Equation 3.12.

$$\begin{aligned}\text{write}_B(\texttt{flag}[B] = \mathit{true}) &\rightarrow \text{write}_B(\texttt{victim} = B) \rightarrow \\ &\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false})\end{aligned} \tag{3.12}$$

It follows that $\text{write}_B(\texttt{flag}[B] = \mathit{true}) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false})$. This observation yields a contradiction because no other write to $\texttt{flag}[B]$ was performed before the critical section executions. ■

Lemma 3.3.4 *The `Peterson` lock algorithm is lockout-free.*

Proof: Suppose not. Suppose (without loss of generality) that A runs forever in the `lock()` method. It must be executing the `while` statement, waiting until either $\texttt{flag}[B]$ becomes false or `victim` is set to B.

What is B doing while A fails to make progress? Perhaps B is repeatedly entering and leaving its critical section. If so, however, then B will set `victim` to B as soon as it reenters the critical section. Once `victim` is set to B, it will not change, and A must eventually return from the `lock()` method, a contradiction.

So it must be that B is also stuck in its call to the `lock()` method, waiting until either $\texttt{flag}[A]$ becomes false or `victim` is set to A. But `victim` cannot be both A and B, a contradiction. ■

Corollary 3.3.5 *The `Peterson` lock algorithm is deadlock-free.*

3.4 The Filter Lock

We now consider two mutual exclusion protocols that work for N threads, where N is greater than 2. The first solution, the `Filter` lock, is a direct generalization of the `Peterson` lock to multiple threads. The second solution, the `Bakery` lock, is perhaps the simplest and best known N-thread solution.

The `Filter` lock, shown in Figure 3.7, creates $N - 1$ "waiting rooms", called *levels*, that a thread must traverse before acquiring the lock. Levels satisfy two important properties:

- At least one thread trying to enter level ℓ succeeds.
- If more than one thread is trying to enter level ℓ, then at least one is blocked.

The `Peterson` lock used a two-element boolean `flag` array to indicate whether a thread is trying to enter the critical section. The `Filter` lock generalizes this notion with an N-element integer `level` array, where the value of `level`$[i]$ indicates the highest level that thread i is interested in entering. Each thread must pass through $N-1$ levels of "exclusion" to enter its critical section. Each level L has a distinct `victim`$[L]$ field used to "filter out" one thread, excluding it from the next level. This array is the natural generalization of the `victim` field in the two-thread algorithm.

We say that a thread A is at *level* 0 if `level`$[A] = 0$. A is at at *level* j for $j > 0$, if A last completed the waiting loop with `level`$[A] \geq j$. (So some thread at level j is also at level $j-1$, and so on.)

Lemma 3.4.1 *For j between 0 and $n-1$, there are at most $n-j$ threads at level j.*

Proof: By induction on j. The base case, where $j = 0$, is trivial.

For the induction step, the induction hypothesis implies that there are at most $n-j+1$ threads at level $j-1$. To show that at least one thread cannot progress to level j, we argue by contradiction: assume there are $n-j+1$ threads at level j.

Let A be the last thread at level j to write to `victim`$[j]$. Because A is last, for any other B at level j:

$$\text{write}_B(\texttt{victim}[j]) \rightarrow \text{write}_A(\texttt{victim}[j]).$$

Inspecting the code, we see that B writes `level`$[B]$ before it writes to `victim`$[j]$:

$$\text{write}_B(\texttt{level}[B] = j) \rightarrow \text{write}_B(\texttt{victim}[j]) \rightarrow \text{write}_A(\texttt{victim}[j]).$$

Inspecting the code, we see that A reads `level`$[B]$ after it writes to `victim`$[j]$.

$$\text{write}_B(\texttt{level}[B] = j) \rightarrow \text{write}_B(\texttt{victim}[j]) \rightarrow \text{write}_A(\texttt{victim}[j]) \rightarrow \text{read}_A(\texttt{level}[B]).$$

Because B is at level j, every time A reads `level`$[B]$, it observes a value greater than or equal to j, implying that A could not have completed its busy-waiting loop, a contradiction. ■

Entering the critical section is equivalent to entering level $n-1$.

```
class Filter implements Lock {
  public static int N; // total number of threads
  private static ThreadLocal<Integer> myIndex;
  private int[] level  = new int[N];
  private int[] victim = new int[N−1];
  public void lock() {
    int i = myIndex.get();
    for (int j = 1; j < N; j++) {
      level [i]  = j;
      victim[j] = i;
      // wait while conflicts exist
      while (sameOrHigher(i,j) && victim[j] == i) {};
    }
  }
  // Is there another thread at the same or higher level?
  private boolean sameOrHigher(int i, int j) {
    for (int k = 0; i < N; k++)
      if (k != i && level[k] >= j)
        return true;
    return false;
  }
  public void unlock() {
    int i = myIndex.get();
    level [i] = 0;
  }
}
```

Figure 3.7: Filter Lock Algorithm

Corollary 3.4.2 *The `Filter` lock algorithm satisfies mutual exclusion.*

Theorem 3.4.3 *The `Filter` lock algorithm is lockout-free.*

Proof: We argue by reverse induction on the levels. The base case, level $n-1$, is trivial, because it contains at most one thread. For the induction hypothesis, assume that every thread that reaches level $j+1$ or higher eventually enters (and leaves) its critical section.

Suppose A is stuck at level j. Eventually, by the induction hypothesis, there will be no threads at higher levels. Once A sets `level`$[A]$ to j, then any thread at level $j-1$ that subsequently reads `level`$[A]$ is prevented from entering level j. Eventually, no more threads enter level j from lower levels. All threads stuck at level j are in the busy-waiting loop, and the values of the `victim` and `level` fields no longer change.

We now argue by induction on the number of threads stuck at level j. For the base case, if A is the only thread at level j or higher, then clearly it will enter level $j+1$. For the induction hypothesis, assume that fewer than k threads cannot be stuck at level j. Suppose threads A and B are stuck at level j. A is stuck as long as it reads `victim`$[j] = A$, and B is stuck as long as it reads `victim`$[j] = B$. The `victim` field is unchanging, and it cannot be equal to both A and B, so one of the two threads will enter level $j+1$, reducing the number of stuck threads to $k-1$, contradicting the induction hypothesis. ■

Corollary 3.4.4 *The `Filter` lock algorithm is deadlock-free.*

3.5 Fairness

The lockout-free property guarantees that every thread that calls `lock()` will eventually enter the critical section, but it makes no guarantees about how long it will take. Ideally (and very informally) if A calls `lock()` before B, then A should enter the critical section before B. Unfortunately, we cannot determine which thread called `lock()` first using the tools at hand. Instead, we split the `lock()` method into two sections of code (with corresponding execution intervals):

1. A *doorway* section, which is *wait-free*, that is, its execution interval D_A consists of a bounded number of steps, and
2. a *waiting* section, whose execution interval W_A may take an unbounded number of steps.

As usual, we use superscripts to indicate repetition.

Here is how we define fairness.

Definition 3.5.1 A lock satisfies the *r-bounded waiting property* if, whenever, thread A finishes its doorway before thread B starts its doorway, then A can be "overtaken" at most r times by B:

If $D_A^j \rightarrow D_B^k$, then $CS_A^j \rightarrow CS_B^{k+r}$.

for threads A and B and integers j and k.

The strong form of fairness known as *first-come-first-served* is equivalent to *0-bounded waiting.*

3.6 Lamport's Bakery Algorithm

The `Bakery` lock algorithm appears in Figure 3.8. It maintains the *first-come-first-served* property by using a distributed version of the "turn-o-matic" machines in often found in bakeries: each thread takes a "number" in the doorway, and then waits until no thread with an earlier number is trying to enter the critical section.

In the `Bakery` lock, `flag`$[A]$ is a Boolean indicating whether A wants to enter the critical section. The `label`$[A]$ field is a label, an integer that indicates the thread's relative order when entering the bakery.

Each time a thread acquires a lock, it generates a new `label` in two steps. First, it reads all the other threads' labels, in some arbitrary order. Second, it generates a label greater by one than the maximal label it read. We call the code from the raising of the flag to the writing of the new `label` the *doorway*. It establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering `<<` on pairs of `label` and thread index: `(label[i],i) << (label[j],j))` if and only if `label[i] < label[j]` or `label[i] == label[j]` and `i < j`.

In the waiting part of the bakery algorithm, a thread repeatedly rereads the labels until it determines that no thread with a raised flag has a lexicographically smaller label/index pair.

Since releasing a thread does not reset the `label`, it is easy to see that each thread's labels are strictly increasing. Interestingly, in both the doorway and waiting sections, threads read the labels asynchronously and in an arbitrary order, so that the set of labels seen prior to picking a new one may have never existed in memory at the same time. Nevertheless, as we will now prove, the algorithm works.

Lemma 3.6.1 *The `Bakery` lock algorithm is deadlock-free.*

Proof: Some waiting thread A has the unique least $($`label`$[A], A)$ pair, and that thread will never wait for another thread. ■

Lemma 3.6.2 *The `Bakery` lock algorithm is first-come-first-served.*

Proof: If A's doorway precedes B's:

$$D_A \rightarrow D_B$$

```
class Bakery implements Lock {
  boolean[] flag = new boolean[N];
  Label[] label = new Label[N];
  static ThreadLocal<Integer> myIndex;
  public void lock() {
    int i = myIndex.get();
     flag [i]   = true;
    label[i] = max(label[0], ..., label[N−1]) + 1;
    while (exists k != i such that
      flag [k] && (label[k],k) << (label[i],i ));
  }
  public void unlock() {
     flag [myIndex.get()] = false;
  }
}
```

Figure 3.8: The Bakery Lock

then A's label is smaller since

$$\text{write}_A(\texttt{label}[A]) \rightarrow \text{read}_B(\texttt{label}[A]) \rightarrow \text{write}_B(\texttt{label}[B]) \rightarrow \text{read}_B(\texttt{flag}[A]),$$

so B is locked out while `flag`$[A]$ is *true*. ■

Note that deadlock freedom and *first-come-first-served* implies lockout freedom.

Lemma 3.6.3 *The* `Bakery` *lock algorithm satisfies mutual exclusion.*

Proof: Suppose not. Let A and B be two threads concurrently in the critical section. Let labeling_A and labeling_B be the last respective sequences of acquiring new labels prior to entering the critical section. Suppose that $(\texttt{label}[A], A) << (\texttt{label}[B], B)$. When B successfully completed the test in its waiting section, it must have read that `flag`$[A]$ was *false* or that $(\texttt{label}[B], B) << (\texttt{label}[A], A)$. However, for a given thread, its index is fixed and its `label` values are strictly increasing, so B must have seen that `flag`$[A]$ was *false*. It follows that

$$\text{labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{labeling}_A$$

which contradicts the assumption that $(\texttt{label}[A], A) << (\texttt{label}[B], B)$. ■

```
public interface Timestamp {
  boolean compare(Timestamp);
}
public class TimestampSystem {
  public Timestamp[] scan();
  public void label(Timestamp timestamp, int i);
}
```

Figure 3.9: A Timestamping System Interface.

3.7 Bounded Timestamps

Notice that the labels of the `Bakery` lock grow without bound, so in a long-lived system we may have to worry about overflow. If a thread's label field silently rolls over from a large number to zero, then the first-come-first-served property no longer holds.

Later on, we will see constructions where counters are used to order threads, or even to produce unique identifiers. How important is the overflow problem in the real world? It is difficult to generalize. Sometimes it matters a great deal. The celebrated "Y2K" bug that captivated the media in the last years of the twentieth century is an example of a genuine overflow problem, even if the consequences were not as dire as predicted. On 18 January 2038, the Unix `time_t` data structure will overflow when the number of seconds since 1 January 1970 exceeds 2^{16}. No one knows whether it will matter. Sometimes, of course, counter overflow is a non-issue. Most applications that use, say, a 64-bit counter are unlikely to last long enough for rollover to occur. (Let the grandchildren worry!)

In the `Bakery` lock, labels act as *timestamps*: they establish an order among the contending threads. Informally, we need to ensure that if one thread takes a label after another, then the latter has the larger label. Inspecting the code for the `Bakery` lock, we see that a thread needs two abilities:

- to read the other threads' timestamps (*scan*), and
- to assign itself a later timestamp (*label*).

A Java interface to such a timestamping system appears in Figure 3.9.

Since our principal application for a bounded timestamping system is to implement the doorway section of the `Lock` class, the timestamping system

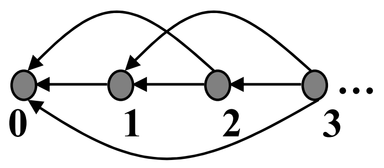

Figure 3.10: The precedence graph for an unbounded timestamping system. The nodes represent the set of all natural numbers and the edges represent the total order among them.

must be wait-free. It is possible to construct such a wait-free *concurrent* timestamping system (see the chapter notes), but the construction is long and rather technical. Instead, we focus on a simpler problem, interesting in its own right: constructing a *sequential* timestamping system, in which threads perform scan and label operations one completely after the other, that is, as if each were performed using mutual exclusion. In other words, consider only executions in which a thread can perform a scan of the other threads' labels, or a scan and then a write of a new label, where each such sequence is a single atomic step. The principles underlying concurrent and sequential timestamping system are essentially the same, but differ substantially in detail.

Think of the range of possible timestamps as nodes of a directed graph (called a *precedence graph*). An edge from node a to node b means that a is a later timestamp than b. The timestamp order is *irreflexive*: there is no edge from any node a to itself. The order is also *antisymmetric*: if there is an edge from a to b, then there is no edge from b to a. Notice that we do *not* require that the order be *transitive*: there can be an edge from a to b and from b to c, without necessarily implying there is an edge from a to c.

Think of assigning a timestamp to a thread as placing that thread's token on that timestamp's node. A thread performs a scan by locating the other threads' tokens, and it assigns itself a new timestamp by moving its own token to a node a such that there is an edge from a to every other thread's node.

Pragmatically, we can implement such a system as an array of single-writer multi-reader fields, where array element A represents the graph node where thread A's most recently placed its token. The `scan()` method takes a "snapshot" of the array, and the `label()` method for thread i updates the i-th array element.

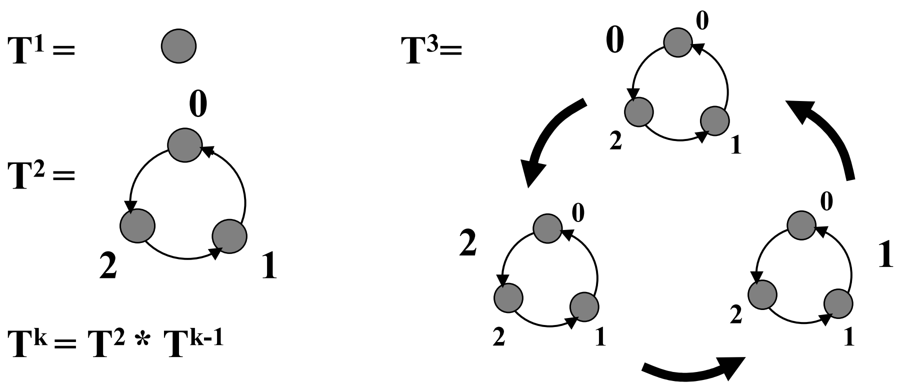

Figure 3.11: The precedence graph for a bounded timestamping system. Consider an initial acyclic situation in which there is a token A on node 12 (node 2 in subgraph 1) and tokens B and C respectively on nodes 21 and 22 (nodes 1 and 2 in subgraph 2). Token B will move to 20 to dominate all others. Token C will then move to 21 to dominate all others, and B and C can continue to cycle in the T^2 subgraph 2 forever. If A is to move to dominate B and C, it cannot pick a node in subgraph 2 since it is full (any T^k subgraph can accommodate at most k tokens). Token A thus moves to node 00. If B now moves, it will choose node 01, C will choose 10 and so on.

Figure 3.10 illustrates the precedence graph for the unbounded timestamp system used in the `Bakery` lock. Not surprisingly, the graph is infinite: there is one node for each natural number, with a directed edge from node a to node b whenever $a > b$.

Consider the precedence graph T^2 shown in Figure 3.11. This graph has three nodes, labeled 0, 1, and 2, and its edges define an ordering relation on the nodes in which 0 is less than 1, 1 is less than 2, and 2 is less than 0. If there are only two threads, then we can use this graph to define a bounded (sequential) timestamping system. The system satisfies the following invariant: the two threads always have tokens located on adjacent nodes, with the direction of the edge indicating their relative order. Suppose A's token is on node 0, and B's token on node 1 (so A has the later timestamp). For A, the `label()` method is trivial: it already has the latest timestamp, so it does nothing. For B, the `label()` method "leapfrogs" A's node by jumping from 0 to 2.

Recall that a *cycle*[1] in a directed graph is a set of nodes $n_0, n_1, \ldots, n_k$ such that there is an edge from n_0 to n_1, from n_1 to n_2, and eventually from n_{k-1} to n_k, and back from n_k to n_0.

The only cycle in the graph T^2 has length three, and there are only two threads, so the order among the threads is never ambiguous. To go beyond two threads, we need additional conceptual tools. Let G be a precedence graph, and A and B subgraphs of G (possibly single nodes). We say that A *dominates* B in G if every node of A has edges directed to every node of B. Let *graph multiplication* be the following composition operator for graphs: $G \circ H$, for graphs G and H, is the following non-commutative operation:

> Replace every node v of G by a copy of H (denoted H_v), and let H_v dominate H_u in $G \circ H$ if v dominates u in G.

Define the graph T^k inductively to be:

1. T^1 is a single node.
2. T^2 is the three-node graph defined above.
3. For $k > 2$, $T^k = T^2 \circ T^{k-1}$.

For example, the graph T^3 is illustrated in Figure 3.11.

The precedence graph T^n is the basis for an N-thread bounded sequential timestamping system. We can "address" any node in the T^n graph with $n-1$ digits, using ternary notation. For example, the nodes in graph T^2 are addressed by 0, 1, and 2. The nodes in graph T^3 are denoted by $00, 01, \ldots, 22$, where the high-order digit indicates one of the three subgraphs, and the low-order digit indicates one node within that subgraph.

The key to understanding the N-thread labeling algorithm is that the nodes covered by tokens can never form a cycle. As mentioned, two threads can never form a cycle on T^2, because the shortest cycle in T^2 requires three nodes.

How does the `label` method work for three threads? When A calls the `label` method, if both of the other threads have tokens on the same T^2 subgraph, then move to a node on the next highest T^2 subgraph, the one whose nodes dominate that T^2 subgraph. For example, consider the graph T^3 as illustrated in Figure 3.11. Assume an initial acyclic situation for in which there is a token A on node 12 (node 2 in subgraph 1) and tokens

[1]The word "cycle" comes from the same Greek root as "circle".

B and C respectively on nodes 21 and 22 (nodes 1 and 2 in subgraph 2). Token B will move to 20 to dominate all others. Token C will then move to 21 to dominate all others, and B and C can continue to cycle in the T^2 subgraph 2 forever. If A is to move to dominate B and C, it cannot pick a node in subgraph 2 since it is full (any T^k subgraph can accommodate at most k tokens). Token A thus moves to node 00. If B now moves, it will choose node 01, C will choose 10 and so on.

3.8 Lower Bounds on Number of Locations

The Bakery lock is succinct, elegant, and fair. So why is it not considered practical? The principal drawback is the need to read N distinct locations, where N is the maximum number of concurrent threads. The number N may be very large, fluctuating, or even unknown. Even worse, threads must be assigned unique indexes between 0 and $n-1$, which is awkward in practice, because threads are often created and destroyed dynamically.

Is there an even cleverer `Lock` algorithm that avoids these problems? There do exist "fast-path" `Lock` algorithms where the number of locations read or written is proportional to the number of threads simultaneously trying to acquire the locks. Nevertheless, we now show that any deadlock-free `Lock` algorithm requires reading or writing at least N distinct locations in the worst case.

Recall that an object's *state* is just the state of its locations. A *global state* is the state of all objects, plus the states of the threads (program counters and local variables).

Definition 3.8.1 A `Lock` object state s is *inconsistent* in any global state where some thread is in the critical section, but the lock state is compatible with a global state in which no thread is in the critical section or is trying to enter.

Lemma 3.8.1 *No deadlock-free `Lock` algorithm can enter an inconsistent state.*

Proof: Suppose the `Lock` object is in an inconsistent state s, where no thread is in the critical section or trying to enter. If thread B tries to enter the critical section, it must eventually succeed, because the algorithm is deadlock-free.

Suppose the `Lock` object is in an inconsistent state s, where A is in the critical section. If thread B tries to enter the critical section, it must block until A leaves.

We have a contradiction, because B cannot determine whether A is in the critical section. ∎

Any `Lock` algorithm that solves deadlock-free mutual exclusion must have N distinct locations. Here, we consider only the three-thread case, showing that a no-deadlock `Lock` algorithm accessed by three threads must have three distinct locations.

Definition 3.8.2 A *covering state* for a `Lock` object is one in which there is at least one thread about to write to each shared location, but the `Lock` object's locations "look" like the critical section is empty (that is, the locations' states appear as if no thread is either in the critical or trying to enter the critical section).

In a covering state, we say that each thread *covers* the location it is about to write.

Theorem 3.8.2 *Any `Lock` algorithm that solves deadlock-free mutual exclusion for three threads must use at least three distinct memory locations.*

Proof: Assume by way of contradiction that we have a no-deadlock `Lock` algorithm for three threads with only two locations. Initially, in state s, no thread is in the critical section or trying to enter. If we run any thread by itself, then it must write to at least one location before entering the critical section, because otherwise s an inconsistent state.

It follows that every thread must write at least one location before entering. If the shared locations are single-writer locations as in the Bakery lock, then it is immediate that three distinct locations are needed.

Now consider multi-writer locations such as the `victim` location in Peterson's algorithm (Figure 3.6). Let s be a covering `Lock` state where A and B respectively cover distinct locations. Consider this possible execution starting from state s:

> Let C run alone. Because the `Lock` algorithm satisfies the no-deadlock property, C will enter the critical section eventually. Then let A and B respectively update their covered locations, leaving the `Lock` object in state s'.

The state s' is inconsistent because no thread can tell whether C is in the critical section, so a lock with two locations is impossible.

It remains to be shown how to maneuver threads A and B into a covering state. Consider an execution in which B runs through the critical section

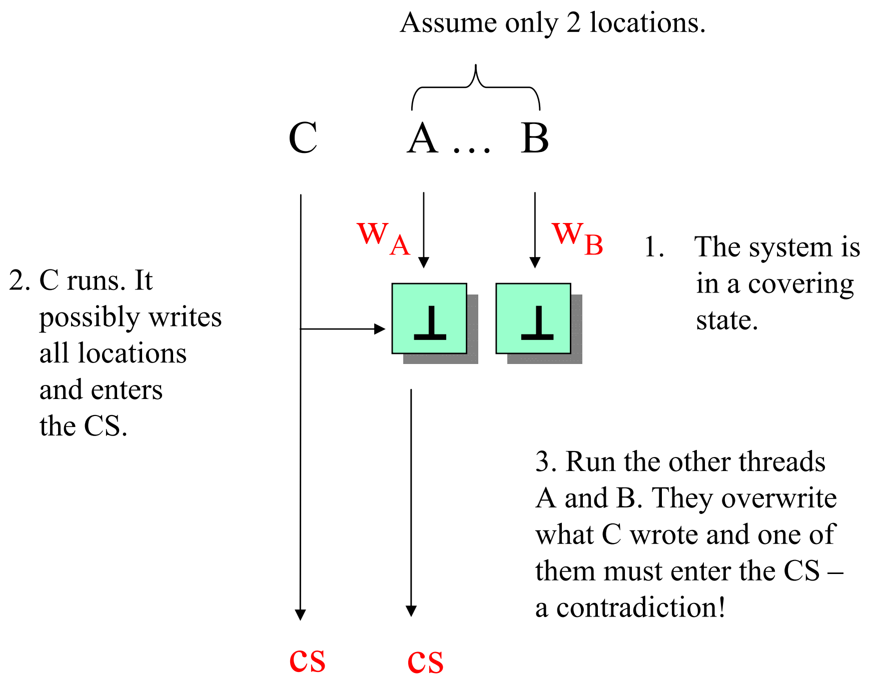
Assume only 2 locations.
C
A … B
W_A
W_B
⊥
⊥
1. The system is in a covering state.
2. C runs. It possibly writes all locations and enters the CS.
3. Run the other threads A and B. They overwrite what C wrote and one of them must enter the CS – a contradiction!
CS
CS

Figure 3.12: Contradiction using a covering state for two locations. Initially both locations have the empty value $\perp$.

three times. Each time around, it must write some location, so consider the first location it writes when trying to enter the critical section. Since there are only two locations, B must write one location twice. Call that location L_B.

Let B run until it is poised to write location L_B for the first time. If A runs now, it would enter the critical section, since B has not written anything. A must write L_A before entering the critical section. Otherwise, if A writes only L_B, then let A enter the critical section, let B write to L_B (obliterating A's last write). The result is an inconsistent state: B cannot tell whether A is in the critical section.

Let A run until it is poised to write L_A. This state is not a covering state, because A could have written something to L_B indicating that it is trying to enter the critical section. Let B run, obliterating any value A might have written to L_B, entering and leaving the critical section at most

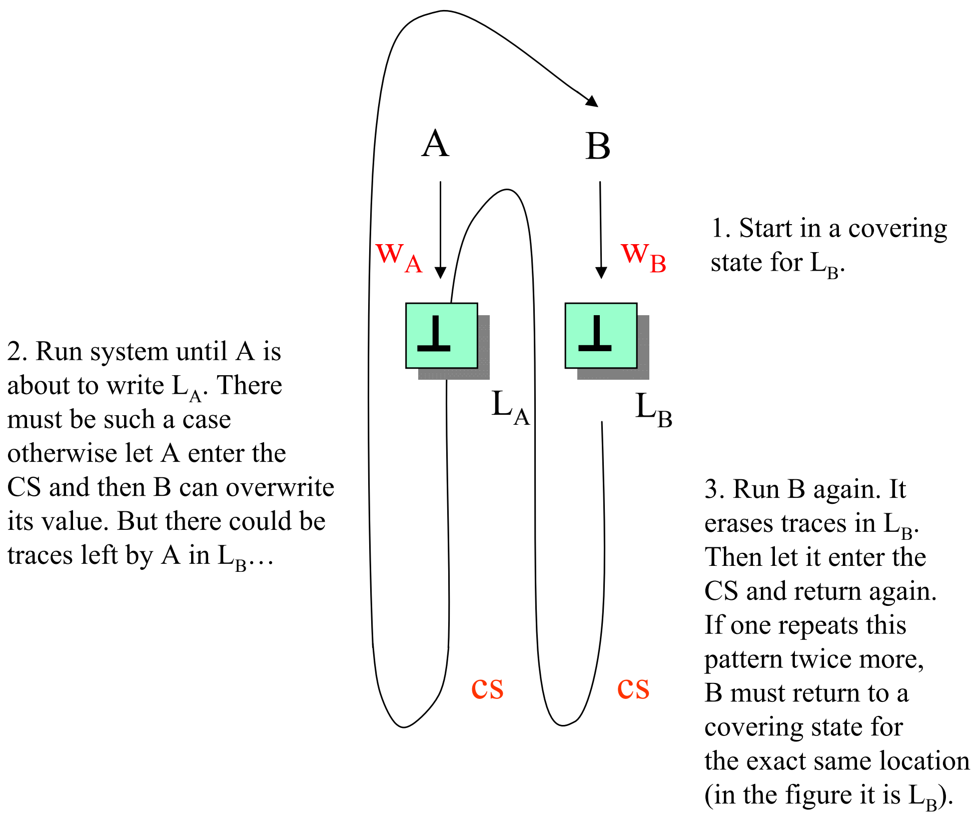
A
B
W_A
W_B
⊥
⊥
L_A
L_B
CS
CS
1. Start in a covering state for L_B.
2. Run system until A is about to write L_A. There must be such a case otherwise let A enter the CS and then B can overwrite its value. But there could be traces left by A in L_B…
3. Run B again. It erases traces in L_B. Then let it enter the CS and return again. If one repeats this pattern twice more, B must return to a covering state for the exact same location (in the figure it is L_B).

Figure 3.13: Reaching a covering state. In the initial covering state for L_B both locations have the empty value $\perp$.

three times, and halting just before its second write to L_B.

In this state, A is about to write L_A, and B about to write L_B, and the locations are consistent with no thread trying or in the critical section, as required in a covering state. Figure 3.13 illustrates this scenario. ■

In later chapters, we will see that modern machine architectures provide specialized instructions for mutual exclusion that allow an N-thread solution using only a constant number of registers. We will also see that making effective use of these instructions is far from trivial.

```
class Queue {
  int head;                    // next item to dequeue
  int tail;                    // next empty slot
  Object[] items = new Object[QSIZE];
  public Queue() {
    head = 0; tail = 0;
  }
  public synchronized void enq(Object x) {
    while (tail - head == QSIZE) {
      try {
        this.wait(); // wait until not full
      } catch (InterruptedException e) {};
    }
    items[tail++ % QSIZE] = x;
    this.notify();
  }
  public Object deq() {
    while (tail - head == QSIZE) {
      try {
        this.wait(); // wait until non-empty
      } catch (InterruptedException e) {};
    }
    Object x = items[head++];
    this.notify();
    return x;
  }
}
```

Figure 3.14: A Lock-based FIFO queue. In Java each object has an implicit lock. A **synchronized** method acquires the lock when called and releases the lock when it returns. The wait method temporarily releases the lock and reacquires it at some later time. The notify method advises waiting threads to reacquire the lock.

3.9 Granularity of Mutual Exclusion

We end this chapter with a discussion of how mutual exclusion can be used in practice. Figure 4.3.1 shows a standard Java implementation of a shared FIFO queue. To understand this code, you must be aware of how Java

```
class LockFreeQueue {
  int head = 0;                    // next item to dequeue
  int tail = 0;                    // next empty slot
  Object[] items = new Object[QSIZE];
  public void enq(Object x) {
    // spin while full
    while (this.tail − this.head == QSIZE) {};
    this.items[this.tail % QSIZE] = x;
    this.tail++;
  }
  public Object deq() {
    // spin while empty
    while (this.tail == this.head) {}; // busy−wait
    Object x = this.items[this.head % QSIZE];
    this.head++;
    return x;
  }
}
```

Figure 3.15: A Lock-free FIFO queue. Initially the `head` and `tail` are equal and the queue is empty. If the `head` and `tail` differ by `QSIZE`, 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.

handles synchronization. Each Java object has an implicit lock field and an implicit condition field. Any method declared to be `synchronized` automatically acquires the lock when the method is called, and releases it when the method returns. If a dequeuing thread discovers the queue is empty, then that thread can wait until something appears in the queue. By calling `this.wait()`, the would-be dequeuer releases the lock and suspends itself. When another thread enqueues an item, it calls `this.notifyAll()` to wake up all suspended threads. These threads compete for the lock, one of them succeeds, and the others go back to waiting.

A key observation about this queue implementation is that every method call locks the entire queue, and so concurrent method calls cannot proceed concurrently. Can we do better? Imagine, for the sake of simplicity, that two threads A and B share a queue, where A always enqueues and B always dequeues. Figure 4.9 shows an implementation of this two-threaded FIFO queue that does not use any locks.

Like its locking-based counterpart, the lock-free queue has three fields:

- `items` is an array of `QSIZE` objects,
- `tail` is the index in the `items` array at which the next enqueued object will be stored; `head` is the index in the `items` array from which the next dequeued object will be removed

If `head` and `tail` differ by `QSIZE`, then the queue is full, and if they are equal, then the queue is empty. The `enq()` method reads the `head` field into a local variable, If the queue is full, the thread *spins*: it repeatedly tests the `tail` field until it observes there is room in the `items` array. It then stores the object in the array, and increments the `tail` field. The enqueue actually "takes effect" when the `tail` field is incremented. The `deq()` method works in a symmetric way.

Note that this implementation does not work if the queue is shared by more than two threads, or if the threads change roles. Later on, we will examine ways in which this example can (and cannot) be generalized.

We contrast these two implementations to emphasize the notion of *granularity* of synchronization. The lock-based queue is an example of *coarse-grained* synchronization: no matter how much support for concurrency the hardware provides, only one thread at a time can execute a method call. The lock-free queue is an example of *fine-grained* synchronization: threads synchronize at the level of individual machine instructions.

Why is this distinction important? There are two reasons. The first is *fault-tolerance.* Recall that modern architectures are asynchronous: a thread can be interrupted at any time for an arbitrary duration (because of cache misses, page faults, descheduling, and so on). If a thread is interrupted while it holds a lock, then all other threads that call that object's methods will also be blocked. The greater the hardware support for concurrency, the greater the wasted resources: the unexpected delay of a single thread can potentially bring a massively parallel computation to a halt.

By contrast, the lock-free queue does not present the same hazards. Threads synchronize at the level of basic machine instructions (reading and updating object fields). The hardware and operating system typically ensure that reading or writing an object field is *atomic*: a thread interrupted while reading or writing a field cannot block other threads attempting to read or write the same field.

The second reason concerns *speedup*. When we reason about the correctness of a multi-threaded program, we do not need to consider the number of physical processors supported by the underlying machine. A single-processor machine can run a multithreaded program as well as a multiprocessor.

Except for performance. Ideally, if we double the number of physical processors, we would like the running time of our programs to be cut in half. This never happens. Realistically, most people who work in this area would be surprised and delighted if, beyond a certain point, doubling the number of processors provided any significant speedup.

To understand why such speedups are difficult, we turn our attention to *Amdahl's Law*. The key idea is that the extent to which we can speed up a program is limited by how much of the program is inherently sequential. The degree to which a program is inherently sequential depends on its granularity of synchronization.

Define the *speedup* S of a program to be the ratio between its running time (measured by a wall clock) on a single-processor machine, and its running time on an N-processor multiprocessor. Let c be the fraction of the program that can be executed concurrently, without synchronization or waiting. If we assume that the sequential program takes time 1, then the sequential part of the program will take time $1-c$, and the concurrent part will take time c/n. Here is the speedup S for an N-processor multiprocessor:

$$S = \frac{1}{1 - c + \frac{c}{n}}$$

For example, if a program spends 20% if its time in critical sections, and is deployed on a 10-processor machine, then Amdahl's Law implies a maximum speedup of

$$3.58 = \frac{1}{1 - 0.8 + \frac{0.8}{10.0}}$$

If we cut the synchronization granularity to 10%, then we have a speedup of

$$5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10.0}}$$

Even small reductions in granularity produce relatively large increases in speedup.

3.10 Exercises

Exercise 3.10.1 Programmers at the Flaky Computer Corporation designed the protocol shown in Figure 3.16 to achieve N-thread mutual exclu-

```
class Flaky implements Lock {
  private int turn;
  private boolean busy = false; // initially false
  // thread−local index, 0 or 1
  private static ThreadLocal<Integer> myIndex;
  public void lock() {
    int i = myIndex.get();
    do {
      do {                         // loop until not busy
        this.turn = i;
      } while (this.busy);
      this.busy = true;
    } while (this.turn != i );
  }
  public void unlock() {
    this.busy = false;
  }
}
```

Figure 3.16: The Flaky Lock used in Exercise 3.10.1.

sion. For each question, either sketch a proof, or display an execution where it fails.

- Does this protocol satisfy mutual exclusion?
- Does this protocol satisfy no-lockout?
- Does this protocol satisfy no-deadlock?

Exercise 3.10.2 Show that the `Filter` class does not provide r-bounded waiting for any finite r.

Exercise 3.10.3 One way to generalize the two-thread Peterson lock is to arrange a number of two-thread Peterson locks in a binary tree. Suppose N is a power of two. Each thread is assigned a leaf lock which it shares with one other thread.

In the tree-lock's `lock()` method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root.

The tree-lock's `unlock()` method for the tree-lock unlocks each of the two-thread Peterson locks that thread has acquired, from the root back to its leaf.

Either sketch a proof that this tree-lock satisfies mutual exclusion, or give an execution where it does not.

Either sketch a proof that this tree-lock satisfies no-lockout, or give an execution where it does not.

Is there an upper bound on the number of times the tree-lock can be acquired and released while a particular thread is trying to acquire the tree-lock?

Exercise 3.10.4 The ℓ-exclusion problem is a variant of the lockout-free mutual exclusion problem. We make two changes: as many as ℓ threads may be in the critical section at the same time, and fewer than ℓ threads might fail (by halting) in the critical section.

Your implementation must satisfy the following conditions:

ℓ-Exclusion: At any time, at most ℓ threads are in the critical section.

ℓ-Lockout-Freedom: As long as fewer than ℓ threads are in the critical section, then some thread that wants to enter the critical section will eventually succeed (even if some threads in the critical section have halted).

Modify Peterson's n-process mutual exclusion algorithm to turn it into an ℓ-exclusion algorithm.

Exercise 3.10.5 In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock.

Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Figure 3.17. They claim that if the base `Lock` class provides mutual exclusion and is livelock-free, so does the `FastPath` lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

Exercise 3.10.6 Suppose N threads call the `visit` method of the `Bouncer` class shown in Figure 3.18. Prove that

```
class FastPath implements Lock {
  private static ThreadLocal<Integer> myIndex;
  private Lock lock;
  private int x, y = −1;
  public void lock() {
    int i = myIndex.get();
    int x = i;                    // I'm here
    while (y != −1) {}            // is the lock free?
    y = i;                        // me again?
    if (x != i)                   // Am I still here?
      lock.lock();                // slow path
  }
  public void unlock() {
    y = −1;
  }
}
```

Figure 3.17: Fast path mutual exclusion algorithm used in Exercise 3.10.5.

- At most one thread gets the value `STOP`.
- At most $N - 1$ threads get the value `DOWN`.
- At most $N - 1$ threads get the value `RIGHT`.

Note that the last two proofs are *not* symmetric.

Exercise 3.10.7 So far, we have assumed that all N threads have unique, small indexes. Here is one way to assign unique small indexes to threads. Arrange `Bouncer` objects in a triangular matrix, where each `Bouncer` is given an id as shown in Figure 3.19. Each thread starts by visiting `Bouncer` zero. If it gets `STOP`, it stops. If it gets `RIGHT`, it visits 1, and if it gets `DOWN`, it visits 2. In general, if a thread gets `STOP`, it stops. If it gets `RIGHT`, it visits the next `Bouncer` on that row, and if it gets `DOWN`, it visits the next `Bouncer` in that column. Each thread takes the id of the `Bouncer` object where it stops.

- Prove that each thread eventually stops at some `Bouncer` object.
- How many `Bouncer` objects will you need in the array if you know in advance the total number n of threads?

```
class Bouncer {
  public static final int DOWN = 0;
  public static final int RIGHT = 1;
  public static final int STOP = 2;
  private boolean goRight = false;
  private ThreadLocal<Integer> myIndex;
  private int last = −1;
  int visit () {
    int i = myIndex.get();
    this.last = i;
    if (this.goRight)
      return RIGHT;
    this.goRight = true;
    if (this.last == i)
      return STOP;
    else
      return DOWN;
  }
}
```

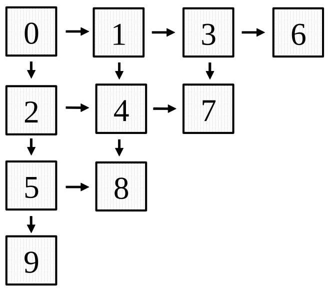

Figure 3.19: Array layout for `Bouncer` objects.

Exercise 3.10.8 Prove, by way of a counterexample, that the sequential time-stamp system T^3, started in a valid state (with no cycles among the labels), will not work for three threads in the concurrent case. Note that it is not a problem to have two identical labels since one can break such ties using thread IDs. The counterexample should display a state of the execution where three labels are not totally ordered.

Exercise 3.10.9 The sequential time-stamp system T^3 had a range of 3^n different possible label values. Design a sequential time-stamp system that requires only $n2^n$ labels. Note that in a time-stamp system, one may look at all the labels in order to choose a new label, yet once a label is chosen, it should be comparable to any other label without knowing what the other labels in the system are. Hint: think of the labels in terms of their bit representation.

Exercise 3.10.10 Give Java code to implement the `Timestamp` interface of Figure 3.9 using unbounded labels. Then, show how to replace the pseudocode of the `Bakery` lock of Figure 3.8 using your `Timestamp` Java code.

Exercise 3.10.11 Running your application on two processors yields a speedup of S_2. Give a formula for S_n, the speedup on n processors, in terms of n and S_2.

Exercise 3.10.12 You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. How would you decide which to buy for a particular application?

3.11 Chapter Notes

Isaac Newton's ideas about the flow of time appear in his famous *Principia* [?]. The "$\rightarrow$" formalism is due to Leslie Lamport [?]. The first three algorithms in this chapter are due to Gary Peterson, who published them in a two-page paper in 1981 [?]. The `Bakery` lock presented here is a simplification of the original Bakery Algorithm due to Leslie Lamport [?]. The sequential timestamp algorithm is due to Amos Israeli and Ming Li [?], who invented the notion of a bounded timestamping system. Danny Dolev and Nir Shavit [?] defined and invented the first bounded concurrent timestamping system. Other bounded timestamping schemes include Sibsankar Haldar and Paul Vitanyi [?], and Cynthia Dwork and Orli Waarts [?]. Our array-based lock-free FIFO queue for a single enqueuer thread and a single dequeuer thread is adapted from Leslie Lamport [?]. The lower bound on the number of lock fields is due to Jim Burns and Nancy Lynch [?]. Their proof technique, called a *covering argument*, has since been widely used to prove lower bounds in distributed computing.