CHAPTER

Mutual exclusion 2

Mutual exclusion is perhaps the most prevalent form of coordination in multiprocessor programming. This chapter covers 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 algorithmic and correctness issues that arise in every area of synchronization. The chapter also provides an impossibility proof. This proof teaches us the limitations of solutions to mutual exclusion that work by reading and writing shared memory, which helps to motivate the real-world mutual exclusion algorithms that appear in later chapters. This chapter is one of the few that contains proofs of algorithms. Though the reader should feel free to skip these proofs, it is helpful to understand the kind of reasoning they present, because we can use the same approach to reason about the practical algorithms considered in later chapters.

2.1 Time and events

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

In 1687, Isaac Newton wrote, "Absolute, True, and Mathematical Time, of itself, and from its own nature flows equably without relation to any thing 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 *instantaneous*: they occur at a single instant of time. It is convenient to require that events are never simultaneous: Distinct events occur at distinct times. (As a practical matter, if we are unsure about the order of two events that happen very close in time, then either order will do.) A thread A produces a sequence of events $a_0, a_1, \ldots$. Programs typically contain loops, so a single program statement can produce many events. One event a *precedes* another event b, written $a \rightarrow b$, if a occurs at an earlier time. The *precedence* relation $\rightarrow$ is a total order on events.

The Art of Multiprocessor Programming. https://doi.org/10.1016/B978-0-12-415950-1.00011-2

Let a_0 and a_1 be events such that $a_0 \rightarrow a_1$. 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 $I_A \rightarrow I_B$, if $a_1 \rightarrow b_0$ (that is, if the final event of I_A precedes the starting event of I_B). The $\rightarrow$ relation is a partial order on intervals. Intervals that are unrelated by $\rightarrow$ are said to be *concurrent*. We also say that an event a *precedes* an interval $I = (b_0, b_1)$, written $a \rightarrow I$, if $a \rightarrow b_0$, and that I *precedes* a, written $I \rightarrow a$, if $b_1 \rightarrow a$.

2.2 Critical sections

In Chapter 1, we discussed the `Counter` class implementation shown in Fig. 2.1. 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 both threads 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 by making these two lines into a *critical 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 achieve mutual exclusion is through a `Lock` object satisfying the interface shown in Fig. 2.2.

```
1  class Counter {
2    private long value;
3    public Counter(long c) {       // constructor
4      value = c;
5    }
6    // increment and return prior value
7    public long getAndIncrement() {
8      long temp = value;          // start of danger zone
9      value = temp + 1;           // end of danger zone
10     return temp;
11   }
12 }
```

FIGURE 2.1

The `Counter` class.

```
1  public interface Lock {
2    public void lock();     // before entering critical section
3    public void unlock();   // before leaving critical section
4  }
```

FIGURE 2.2

The `Lock` interface.

```
1  public class Counter {
2    private long value;
3    private Lock lock;              // to protect critical section
4
5    public long getAndIncrement() {
6      lock.lock();                  // enter critical section
7      try {
8        long temp = value;          // in critical section
9        value = temp + 1;           // in critical section
10       return temp;
11     } finally {
12       lock.unlock();              // leave critical section
13     }
14   }
15 }
```

FIGURE 2.3

Using a lock object.

Fig. 2.3 shows how to use a `Lock` object 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 `Lock` object,
2. the thread calls that object's `lock()` method when it wants to enter the critical section, and
3. the thread calls the `unlock()` method when it leaves the critical section.

PRAGMA 2.2.1

In Java, the `lock()` and `unlock()` methods should be used in the following structured way:

```
1  mutex.lock();
2  try {
3    ...           // body
4  } finally {
5    ... // restore invariant if needed
6    mutex.unlock();
7  }
```

This idiom ensures that the lock is acquired before entering the **`try`** block, and that the lock is released when control leaves the block. If a statement in the block throws an unexpected exception, it may be necessary to restore the object to a consistent state before returning.

We say that a thread *acquires* (alternatively, *locks*) a lock when it returns from a `lock()` method call, and *releases* (alternatively, *unlocks*) the lock when it invokes the `unlock()` method. If a thread has acquired and not subsequently released a lock, we say that the thread *holds* the lock. No thread may acquire the lock while any other thread holds it, so at most one thread holds the lock at any time. We say the lock is *busy* if a thread holds it; otherwise, we say the lock is *free*.

Multiple critical sections may be associated with the same `Lock`, in which case no thread may execute a critical section while any other thread is executing a critical section associated with the same `Lock`. From the perspective of a `Lock` algorithm, a thread starts a critical section when its call to the `lock()` method returns, and it ends the critical section by invoking the `unlock()` method; that is, a thread executes the critical section while it holds the lock.

We now state more precisely the properties that a good `Lock` algorithm should satisfy, assuming that every thread that acquires the lock eventually releases it.

Mutual exclusion At most one thread holds the lock at any time.

Freedom from deadlock If a thread is attempting to acquire or release the lock (i.e., it invoked `lock()` or `unlock()` and has not returned), then eventually some thread acquires or releases the lock (i.e., it returns from invoking `lock()` or `unlock()`). If a thread calls `lock()` and never returns, then other threads must complete an infinite number of critical sections.

Freedom from starvation Every thread that attempts to acquire or release the lock eventually succeeds (i.e., every call to `lock()` or `unlock()` eventually returns).

Note that starvation-freedom implies deadlock-freedom.

The mutual exclusion property is clearly essential. It guarantees that the critical section, that is, the code executed between the acquisition and release of the lock, is executed by at most one thread at a time. In other words, executions of the critical section cannot overlap. Without this property, we cannot guarantee that a computation's results are correct.

Let CS_A^j be the interval during which thread A executes the critical section for the jth time. Thus, $CS_A^j = (a_0, a_1)$, where a_0 is the response event for A's jth call to `lock()` and a_1 is the invocation event for A's jth call to `unlock()`. For two distinct threads A and B and integers j and k, either $CS_A^j \rightarrow CS_B^k$ or $CS_B^k \rightarrow CS_A^j$.

The deadlock-freedom property is important. It implies that the system never "freezes." If some thread calls `lock()` and never acquires the lock, then either some other thread acquires and never releases the lock or other threads must be completing an infinite number of critical sections. Individual threads may be stuck forever (called *starvation*), but some thread makes progress.

The starvation-freedom property, while clearly desirable, is the least compelling of the three. This property is sometimes called *lockout-freedom*. In later chapters, we discuss practical mutual exclusion algorithms that are not starvation-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 starvation-freedom property is also weak in the sense that there is no guarantee for how long a thread waits before it enters the critical section. In later chapters, we look at algorithms that place bounds on how long a thread can wait.

In the terminology of Chapter 1, mutual exclusion is a safety property, and deadlock-freedom and starvation-freedom are liveness properties.

2.3 Two-thread solutions

We now consider algorithms that solve the mutual exclusion problem for two threads. Our two-thread lock algorithms follow the following conventions: The threads have IDs 0 and 1, and a thread can acquire its ID by calling `ThreadID.get()`. We store the calling thread's ID in i and the other thread's ID in $j = 1 - i$.

We begin with two inadequate but interesting algorithms.

2.3.1 The `LockOne` class

Fig. 2.4 shows the `LockOne` algorithm, which maintains a Boolean `flag` variable for each thread. To acquire the lock, a thread sets its `flag` to *true* and waits until the other thread's `flag` is *false*. The thread releases the flag by setting its `flag` back to *false*.

We use $\text{write}_A(x = v)$ to denote the event in which thread A assigns value v to field x, and $\text{read}_A(x == v)$ to denote the event in which A reads v from field x. For example, in Fig. 2.4, the event $\text{write}_A(\texttt{flag}[i] = \textit{true})$ is caused by line 7 of the `lock()` method. We sometimes omit the value when it is unimportant.

Lemma 2.3.1. The `LockOne` algorithm satisfies mutual exclusion.

```
1  class LockOne implements Lock {
2    private boolean[] flag = new boolean[2];
3    // thread-local index, 0 or 1
4    public void lock() {
5      int i = ThreadID.get();
6      int j = 1 - i;
7      flag[i] = true;
8      while (flag[j]) {}        // wait until flag[j] == false
9    }
10   public void unlock() {
11     int i = ThreadID.get();
12     flag[i] = false;
13   }
14 }
```

FIGURE 2.4

Pseudocode for the `LockOne` algorithm.

PRAGMA 2.3.1

In practice, the Boolean `flag` variables in Fig. 2.4, as well as the `victim` and `label` variables in later algorithms, must all be declared **`volatile`** to work properly. We explain the reasons in Chapter 3 and Appendix B. For readability, we omit the **`volatile`** declarations for now. We begin declaring the appropriate variables as volatile in Chapter 7.

Proof. Suppose not. Then there are overlapping critical sections CS_A and CS_B of threads A and B, respectively ($A \neq B$). Consider each thread's last execution of the `lock()` method before entering its critical section. Inspecting the code, we see that

$$\text{write}_A(\texttt{flag}[A] = \mathit{true}) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false}) \rightarrow CS_A,$$
$$\text{write}_B(\texttt{flag}[B] = \mathit{true}) \rightarrow \text{read}_B(\texttt{flag}[A] == \mathit{false}) \rightarrow CS_B.$$

Note that once `flag`$[B]$ is set to *true* it remains *true* until B exits its critical section. Since the critical sections overlap, A must read `flag`$[B]$ before B sets it to *true*. Similarly, B must read `flag`$[A]$ before A sets it to *true*. Combining these, we get

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

There is a cycle in $\rightarrow$, which is a contradiction, because $\rightarrow$ is a partial order (an event cannot precede itself). □

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

2.3.2 The `LockTwo` class

Fig. 2.5 shows an alternative lock algorithm, the `LockTwo` class, which uses a single `victim` field that indicates which thread should yield. To acquire the lock, a thread sets the `victim` field to its own ID and then waits until the other thread changes it.

Lemma 2.3.2. The `LockTwo` algorithm satisfies mutual exclusion.

Proof. Suppose not. Then there are overlapping critical sections CS_A and CS_B of threads A and B, respectively ($A \neq B$). As before, consider each thread's last execution of the `lock()` method before entering its critical section. Inspecting the code, we see that

```
1  class LockTwo implements Lock {
2    private int victim;
3    public void lock() {
4      int i = ThreadID.get();
5      victim = i;              // let the other go first
6      while (victim == i) {}   // wait
7    }
8    public void unlock() {}
9  }
```

FIGURE 2.5

Pseudocode for the `LockTwo` algorithm.

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{victim} == B) \rightarrow CS_A,$$
$$\text{write}_B(\texttt{victim} = B) \rightarrow \text{read}_B(\texttt{victim} == A) \rightarrow CS_B.$$

Thread B must assign B to the `victim` field between events $\text{write}_A(\texttt{victim} = A)$ and $\text{read}_A(\texttt{victim} = B)$, so in particular, B must write `victim` after A. However, by the same reasoning, A must write `victim` after B, which is a contradiction. □

The `LockTwo` algorithm is inadequate because it gets stuck *unless* the threads run concurrently. Nevertheless, `LockTwo` has 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 get stuck.

2.3.3 The Peterson lock

We combine the `LockOne` and `LockTwo` algorithms to construct a starvation-free lock algorithm, shown in Fig. 2.6. This algorithm—known as *Peterson's algorithm*, after its inventor—is arguably the most succinct and elegant two-thread mutual exclusion algorithm.

Lemma 2.3.3. 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 before overlapping critical sections CS_A and CS_B. Inspecting the code, we see that

$$\begin{aligned} &\text{write}_A(\texttt{flag}[A] = \textit{true}) \rightarrow \text{write}_A(\texttt{victim} = A) \qquad (2.3.1)\\ &\quad \rightarrow \text{read}_A(\texttt{flag}[B]) \rightarrow \text{read}_A(\texttt{victim}) \rightarrow CS_A,\\ &\text{write}_B(\texttt{flag}[B] = \textit{true}) \rightarrow \text{write}_B(\texttt{victim} = B) \qquad (2.3.2)\\ &\quad \rightarrow \text{read}_B(\texttt{flag}[A]) \rightarrow \text{read}_B(\texttt{victim}) \rightarrow CS_B. \end{aligned}$$

```
1  class Peterson implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    private int victim;
5    public void lock() {
6      int i = ThreadID.get();
7      int j = 1 - i;
8      flag[i] = true;            // I'm interested
9      victim = i;                // you go first
10     while (flag[j] && victim == i) {} // wait
11   }
12   public void unlock() {
13     int i = ThreadID.get();
14     flag[i] = false;           // I'm not interested
15   }
16 }
```

FIGURE 2.6

Pseudocode for the `Peterson` lock algorithm.

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

$$\text{write}_B(\texttt{victim} = B) \rightarrow \text{write}_A(\texttt{victim} = A). \tag{2.3.3}$$

Eq. (2.3.3) implies that A observed `victim` to be A in Eq. (2.3.1). Since A nevertheless entered its critical section, it must have observed `flag`$[B]$ to be *false*, so we have

$$\text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false}). \tag{2.3.4}$$

Putting Eqs. (2.3.2) to (2.3.4) together yields:

$$\begin{aligned} &\text{write}_B(\texttt{flag}[B] = \mathit{true}) \rightarrow \text{write}_B(\texttt{victim} = B) \\ &\quad \rightarrow \text{write}_A(\texttt{victim} = A) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false}). \end{aligned} \tag{2.3.5}$$

By the transitivity of $\rightarrow$, $\text{write}_B(\texttt{flag}[B] = \mathit{true}) \rightarrow \text{read}_A(\texttt{flag}[B] == \mathit{false})$. This observation yields a contradiction because no other write to `flag`$[B]$ was performed before the critical section executions. □

Lemma 2.3.4. The `Peterson` lock algorithm is starvation-free.

Proof. Suppose not, so some thread runs forever in the `lock()` method. Suppose (without loss of generality) that it is A. It must be executing the **`while`** statement, waiting until either `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 sets `victim` to B before it

reenters the critical section. Once `victim` is set to B, it does not change, and A must eventually return from the `lock()` method, a contradiction.

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

Corollary 2.3.5. The `Peterson` lock algorithm is deadlock-free.

2.4 Notes on deadlock

Although the `Peterson` lock algorithm is deadlock-free (and even starvation-free), another kind of deadlock can arise in programs that use multiple `Peterson` locks (or any other lock implementation). For example, suppose that threads A and B share locks ℓ_0 and ℓ_1, and that A acquires ℓ_0 and B acquires ℓ_1. If A then tries to acquire ℓ_1 and B tries to acquire ℓ_0, the threads deadlock because each one waits for the other to release its lock.

In the literature, the term *deadlock* is sometimes used more narrowly to refer to the case in which the system enters a state from which there is no way for threads to make progress. The `LockOne` and `LockTwo` algorithms are susceptible to this kind of deadlock: In both algorithms, both threads can get stuck waiting in their respective while loops.

This narrower notion of deadlock is distinguished from *livelock*, in which two or more threads actively prevent each other from making progress by taking steps that subvert steps taken by other threads. When the system is livelocked rather than deadlocked, there is some way to schedule the threads so that the system can make progress (but also some way to schedule them so that there is no progress). Our definition of deadlock-freedom proscribes livelock as well as this narrower notion of deadlock.

Consider, for example, the `Livelock` algorithm in Fig. 2.7. (This is a variant of the flag protocol described in Section 1.2 in which both threads follow Bob's part of the protocol.) If both threads execute the `lock()` method, they may indefinitely repeat the following steps:

- Set their respective `flag` variables to *true*.
- See that the other thread's `flag` is *true*.
- Set their respective `flag` variables to *false*.
- See that the other thread's `flag` is *false*.

Because of this possible livelock, `Livelock` is not deadlock-free by our definition.

However, `Livelock` does not deadlock by the narrower definition because there is always some way to schedule the threads so that one of them makes progress: If one thread's `flag` is *false*, then execute the other thread until it exits the loop and returns. If both threads' `flag` variables are *true*, then execute one thread until it sets its `flag` to *false*, and then execute the other as described above (i.e., until it returns).

```
1  class Livelock implements Lock {
2    // thread-local index, 0 or 1
3    private boolean[] flag = new boolean[2];
4    public void lock() {
5      int i = ThreadID.get();
6      int j = 1 - i;
7      flag[i] = true;
8      while (flag[j]) {
9        flag[i] = false;
10       while (flag[j]) {}        // wait
11       flag[i] = true;
12     }
13   }
14   public void unlock() {
15     int i = ThreadID.get();
16     flag[i] = false;
17   }
18 }
```

FIGURE 2.7

Pseudocode for a lock algorithm that may livelock.

2.5 The filter lock

The `Filter` lock, shown in Fig. 2.8, generalizes the `Peterson` lock to work for $n > 2$ threads. It creates $n - 1$ "waiting rooms," called *levels*, that a thread must traverse before acquiring the lock. The levels are depicted in Fig. 2.9. 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 (i.e., continues to wait without entering that level).

The `Peterson` lock uses 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`[A] indicates the highest level that thread A is trying to enter. 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 that level unless no thread is at that level or higher.

Initially, a thread A is at level 0. A *enters* level ℓ for $\ell > 0$ when it completes the waiting loop on line 17 with `level`[A] $= \ell$ (i.e., when it stops waiting at that loop). A enters its critical section when it enters level $n - 1$. When A leaves the critical section, it sets `level`[A] to 0.

```
1  class Filter implements Lock {
2    int[] level;
3    int[] victim;
4    public Filter(int n) {
5      level = new int[n];
6      victim = new int[n]; // use 1..n-1
7      for (int i = 0; i < n; i++) {
8        level[i] = 0;
9      }
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     for (int i = 1; i < n; i++) { // attempt to enter level i
14       level[me] = i;
15       victim[i] = me;
16       // spin while conflicts exist
17       while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18     }
19   }
20   public void unlock() {
21     int me = ThreadID.get();
22     level[me] = 0;
23   }
24 }
```

FIGURE 2.8

Pseudocode for the `Filter` lock algorithm.

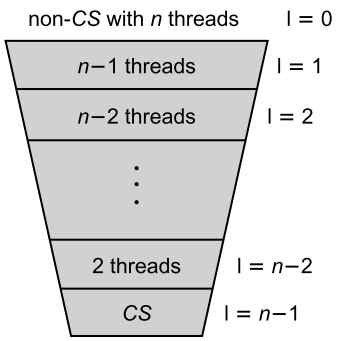

FIGURE 2.9

Threads pass through $n-1$ levels, the last of which is the critical section. Initially, all n threads are at level 0. At most $n-1$ enter level 1, at most $n-2$ enter level 2, and so on, so that only one thread enters the critical section at level $n-1$.

Lemma 2.5.1. For j between 0 and $n-1$, at most $n-j$ threads have entered level j (and not subsequently exited the critical section).

Proof. We prove this by induction on j. The base case, where $j = 0$, is trivial. For the induction step, the induction hypothesis implies that at most $n-j+1$ threads have entered level $j-1$. To show that at least one thread does not enter level j, we argue by contradiction. Assume that $n-j+1$ threads have entered level j. Because $j \leq n-1$, there must be at least two such threads.

Let A be the last thread to write `victim`$[j]$. A must have entered level j since `victim`$[j]$ is written only by threads that have entered level $j-1$, and, by the induction hypothesis, every thread that has entered level $j-1$ has also entered level j. Let B be any thread other than A that has entered level j. Inspecting the code, we see that before B enters level j, it first writes j to `level`$[B]$ and then writes B to `victim`$[j]$. Since A is the last to write `victim`$[j]$, we have

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

We also see that A reads `level`$[B]$ (line 17) after it writes to `victim`$[j]$, so

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

Because B has entered level j, every time A reads `level`$[B]$, it observes a value greater than or equal to j, and since `victim`$[j] = A$ (because A was the last to write it), A could not have completed its waiting loop on line 17, a contradiction. □

Corollary 2.5.2. The `Filter` lock algorithm satisfies mutual exclusion.

Proof. Entering the critical section is equivalent to entering level $n-1$, so at most one thread enters the critical section. □

Lemma 2.5.3. The `Filter` lock algorithm is starvation-free.

Proof. We prove by induction on j that every thread that enters level $n-j$ eventually enters and leaves the critical section (assuming that it keeps taking steps and that every thread that enters the critical section eventually leaves it). The base case, with $j = 1$, is trivial because level $n-1$ is the critical section.

For the induction step, we suppose that every thread that enters level $n-j$ or higher eventually enters and leaves the critical section, and show that every thread that enters level $n-j-1$ does too.

Suppose, for contradiction, that a thread A has entered level $n-j-1$ and is stuck. By the induction hypothesis, it never enters level $n-j$, so it must be stuck at line 17 with `level`$[A] = n-j$ and `victim`$[n-j] = A$. After A writes `victim`$[n-j]$, no thread enters level $n-j-1$ because any thread that did would overwrite `victim`$[n-j]$, allowing A to enter level $n-j$. Furthermore, any other thread B trying to enter level $n-j$ will eventually succeed because `victim`$[n-j] = A \neq B$,

so eventually no threads other than A are trying to enter level $n - j$. Moreover, any thread that enters level $n - j$ will, by the induction hypothesis, enter and leave the critical section, setting its level to 0. At some point, A is the only thread that has entered level $n - j - 1$ and not entered and left the critical section. In particular, after this point, `level`$[B] < n - j$ for every thread B other than A, so A can enter level $n - j$, a contradiction. □

Corollary 2.5.4. The `Filter` lock algorithm is deadlock-free.

2.6 Fairness

The starvation-freedom property guarantees that every thread that calls `lock()` eventually enters the critical section, but it makes no guarantees about how long this may take, nor does it guarantee that the lock will be "fair" to the threads attempting to acquire it. For example, although the `Filter` lock algorithm is starvation-free, a thread attempting to acquire the lock may be overtaken arbitrarily many times by another thread.

Ideally (and very informally), if A calls `lock()` before B, then A should enter the critical section before B. That is, the lock should be "first-come-first-served." However, with the tools we have introduced so far, we cannot determine which thread called `lock()` first.

To define fairness, we split the `lock()` method into a *doorway* section and a *waiting* section, where the doorway section always completes in a bounded number of steps (the waiting section may take an unbounded number of steps). That is, there is a fixed limit on the number of steps a thread may take after invoking `lock()` before it completes the doorway section.

A section of code that is guaranteed to complete in a bounded number of steps is said to be *bounded wait-free*. The bounded wait-free property is a strong progress requirement. It is satisfied by code that has no loops. In later chapters, we discuss ways to provide this property in code that has loops. With this definition, we define the following fairness property.

Definition 2.6.1. A lock is *first-come-first-served* if its `lock()` method can be split into a bounded wait-free doorway section followed by a waiting section so that whenever thread A finishes its doorway before thread B starts its doorway, A cannot be overtaken by B. That is,

$$\text{if } D_A^j \rightarrow D_B^k \text{ then } CS_A^j \rightarrow CS_B^k$$

for any threads A and B and integers j and k, where D_A^j and CS_A^j are the intervals during which A executes the doorway section of its jth call to the `lock()` method and its jth critical section, respectively.

Note that any algorithm that is both deadlock-free and first-come-first-served is also starvation-free.

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5      flag = new boolean[n];
6      label = new Label[n];
7      for (int i = 0; i < n; i++) {
8        flag[i] = false; label[i] = 0;
9      }
10   }
11   public void lock() {
12     int i = ThreadID.get();
13     flag[i] = true;
14     label[i] = max(label[0], ...,label[n-1]) + 1;
15     while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }
17   public void unlock() {
18     flag[ThreadID.get()] = false;
19   }
20 }
```

FIGURE 2.10

Pseudocode for the `Bakery` lock algorithm.

2.7 Lamport's Bakery algorithm

Perhaps the most elegant solution to the mutual exclusion problem for n threads is the `Bakery` lock algorithm, which appears in Fig. 2.10. It guarantees the *first-come-first-served* property by using a distributed version of the number-dispensing machines 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 flag that indicates whether A wants to enter the critical section, and label$[A]$ is an integer that indicates the thread's relative order when entering the bakery, for each thread A. To acquire the lock, a thread first raises its flag, and then picks a new label by reading the labels of all the threads (in any order) and generating a label greater than all the labels it read. The code from the invocation of the `lock()` method to the writing of the new label (line 14) is the *doorway*: it establishes that thread's order with respect to other threads trying to acquire the lock. Threads that execute their doorways concurrently may read the same labels and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering $\ll$ on pairs of labels and thread IDs:

$$\begin{gathered}(\mathsf{label}[i], i) \ll (\mathsf{label}[j], j)) \\ \text{if and only if} \\ \mathsf{label}[i] < \mathsf{label}[j] \quad \text{or} \quad \mathsf{label}[i] = \mathsf{label}[j] \quad \text{and} \quad i < j.\end{gathered} \tag{2.7.1}$$

In the waiting section of the Bakery algorithm (line 15), a thread repeatedly reads the flags and labels of the other threads in any order until it determines that no thread with a raised flag has a lexicographically smaller label/ID pair.

Since releasing a lock 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. For example, the set of labels seen prior to picking a new one may have never existed in memory at the same time. Nevertheless, the algorithm works.

Lemma 2.7.1. The Bakery lock algorithm is deadlock-free.

Proof. Some waiting thread A has the unique least $(\mathsf{label}[A], A)$ pair, and that thread never waits for another thread. □

Lemma 2.7.2. The Bakery lock algorithm is first-come-first-served.

Proof. If A's doorway precedes B's, then A's label is smaller since

$$\mathrm{write}_A(\mathsf{label}[A]) \to \mathrm{read}_B(\mathsf{label}[A]) \to \mathrm{write}_B(\mathsf{label}[B]) \to \mathrm{read}_B(\mathtt{flag}[A]),$$

so B is locked out while $\mathtt{flag}[A]$ is *true*. □

Corollary 2.7.3. The Bakery lock algorithm is starvation-free.

Proof. This follows immediately from Lemmas 2.7.1 and 2.7.2 because any deadlock-free first-come-first-served lock algorithm is also starvation-free. □

Lemma 2.7.4. The Bakery lock algorithm satisfies mutual exclusion.

Proof. Suppose not. Let A and B be two threads concurrently in the critical section with $(\mathsf{label}[A], A) \ll (\mathsf{label}[B], B)$. Let $\mathrm{labeling}_A$ and $\mathrm{labeling}_B$ be the last respective sequences of acquiring new labels (line 14) prior to entering the critical section. To complete its waiting section, B must have read either that $\mathtt{flag}[A]$ was *false* or that $(\mathsf{label}[B], B) \ll (\mathsf{label}[A], A)$. However, for a given thread, its ID is fixed and its label[] values are strictly increasing, so B must have seen that $\mathtt{flag}[A]$ was *false*. It follows that

$$\mathrm{labeling}_B \to \mathrm{read}_B(\mathrm{flag}[A] == \mathit{false}) \to \mathrm{write}_A(\mathrm{flag}[A] = \mathit{true}) \to \mathrm{labeling}_A,$$

which contradicts the assumption that $(\mathsf{label}[A], A) \ll (\mathsf{label}[B], B)$. □

2.8 Bounded timestamps

Note 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.

In later chapters, we discuss constructions in which counters are used to order threads, or even to produce unique IDs. 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 20th century is an example of a genuine overflow problem, even if the consequences were not as dire as predicted. On January 19, 2038, the Unix `time_t` data structure will overflow when the number of seconds since January 1, 1970 exceeds 2^{31}. No one knows whether it will matter. Sometimes, of course, counter overflow is a nonissue. Most applications that use, say, a 64-bit counter are unlikely to last long enough for roll-over 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 Fig. 2.11. Since our principal application for a bounded timestamping system is to implement the doorway section of the `Lock` class, the timestamping system 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 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, we consider only executions in which a thread can perform a scan of the other threads' labels, or a scan and then an assignment of a new label, where each such sequence is a single atomic step. Although the details of concurrent and sequential timestamping systems differ substantially, the principles underlying them are essentially the same.

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

```
1  public interface Timestamp {
2    boolean compare(Timestamp);
3  }
4  public interface TimestampSystem {
5    public Timestamp[] scan();
6    public void label(Timestamp timestamp, int i);
7  }
```

FIGURE 2.11

A timestamping system interface.

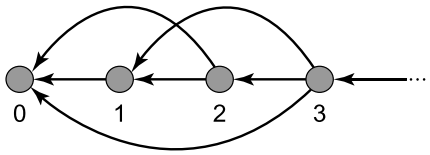

FIGURE 2.12

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.

from b to a. 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, with an element for each thread A that indicates the node that A most recently assigned its token. The `scan()` method takes a "snapshot" of the array, and the `label()` method for thread A updates the array element for A.

Fig. 2.12 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 Fig. 2.13. 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 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 B has the later timestamp). For B, the `label()` method is trivial: It already has the latest timestamp, so it does nothing. For A, the `label()` method "leapfrogs" B's node by jumping from 0 to 2.

Recall that a *cycle* 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 noncommutative composition operation for graphs (denoted $G \circ H$):

> 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.

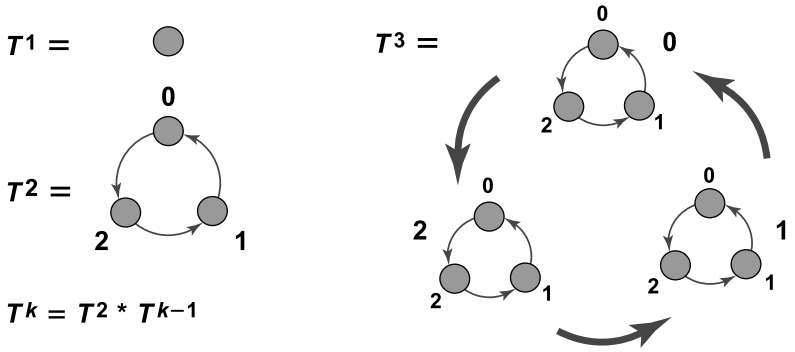

FIGURE 2.13

The precedence graph for a bounded timestamping system. Consider an initial situation in which there is a token *A* on node 12 (node 2 in subgraph 1) and tokens *B* and *C* on nodes 21 and 22 (nodes 1 and 2 in subgraph 2). Token *B* will move to node 20 to dominate the others. Token *C* will then move to node 21 to dominate the 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). Instead, token *A* moves to node 00. If *B* now moves, it will choose node 01, *C* will choose node 10, and so on.

Define the graph T^k inductively as follows:

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

For example, the graph T^3 is illustrated in Fig. 2.13.

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 `label()`, if both 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 Fig. 2.13. We assume an initial acyclic situation in which there is a token *A* on node 12 (node 2 in subgraph 1) and tokens *B* and *C* on nodes 21 and 22 (nodes 1 and 2 in subgraph 2). Token *B* will move to node 20 to dominate all others. Token *C* will then move to node 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 node 10, and so on.

2.9 Lower bounds on the 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 and write n distinct locations, where n is the maximum number of concurrent threads (which may be very large).

Is there a clever `Lock` algorithm based on reading and writing memory that avoids this overhead? We now demonstrate that the answer is *no*. Any deadlock-free mutual exclusion algorithm requires allocating and then reading or writing at least n distinct locations in the worst case. This result is crucial: it motivates us to augment our multiprocessor machines with synchronization operations stronger than reads and writes, and use them as the basis of our mutual exclusion algorithms. We discuss practical mutual exclusion algorithms in later chapters.

In this section, we examine why this linear bound is inherent. We observe the following limitation of memory accessed solely by *read* or *write* instructions (typically called *loads* and *stores*): Information written by a thread to a given location may be *overwritten* (i.e., stored to) without any other thread ever seeing it.

Our proof requires us to argue about the state of all memory used by a given multithreaded program. An object's *state* is just the state of its fields. A thread's *local state* is the state of its program counters and local variables. A *global state* or *system state* is the state of all objects, plus the local states of the threads.

Definition 2.9.1. 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 the critical section.

Lemma 2.9.2. No deadlock-free `Lock` algorithm can enter an inconsistent state.

Proof. Suppose the `Lock` object is in an inconsistent state s, where some thread A is in the critical section. If thread B tries to enter the critical section, it must eventually succeed because the algorithm is deadlock-free and B cannot determine that A is in the critical section, a contradiction. □

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 deadlock-free `Lock` algorithm accessed by three threads must use three distinct locations.

Definition 2.9.3. 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 (i.e., the locations' states appear as if there is no thread either in the critical section 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 2.9.4. Any `Lock` algorithm that, by reading and writing memory, 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 deadlock-free `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, as otherwise it creates 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 multiwriter locations such as elements of the `victim` array in Peterson's algorithm (Fig. 2.6). Assume that we can bring the system to a covering `Lock` state s where A and B cover distinct locations. Consider the following possible execution starting from state s as depicted in Fig. 2.14:

> Let C run alone. Because the `Lock` algorithm satisfies the deadlock-free property, C enters the critical section eventually. Then let A and B respectively update their covered locations, leaving the `Lock` object in state s'.

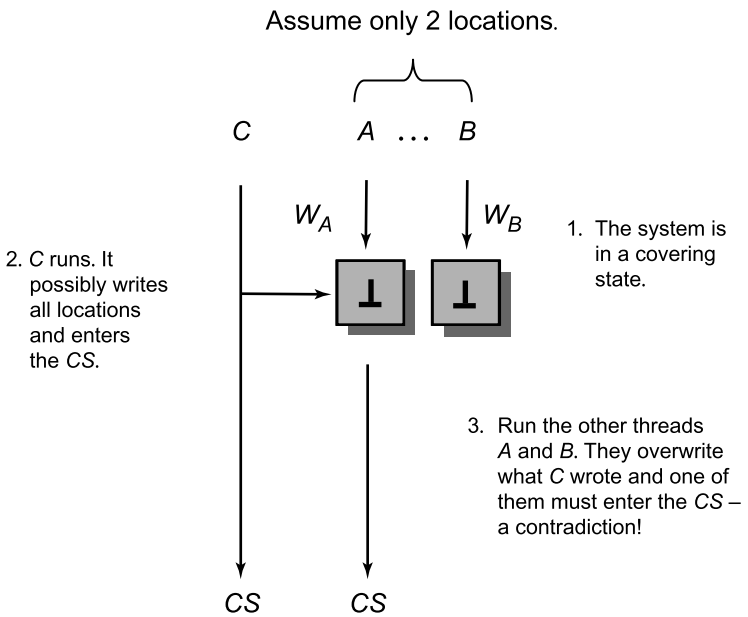

FIGURE 2.14

Contradiction using a covering state for two locations. Initially both locations have the empty value ⊥.

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

It remains to show how to maneuver threads A and B into a covering state. Consider an execution in which B runs through the critical section three times. Each time around, it must write to some location, so consider the first location to which it writes when trying to enter the critical section. Since there are only two locations, B must "write first" to the same location twice. Call that location L_B.

Let B run until it is poised to write to 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 to L_A before entering the critical section. Otherwise, if A writes only to L_B, then let A enter the critical section, and 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 to L_A. This state might not be a covering state because A could have written something to L_B indicating to thread C 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 three times, and halting just before its second write to L_B. Note that every time B enters and leaves the critical section, whatever it wrote to the locations is no longer relevant.

In this state, A is about to write to L_A, B is about to write to L_B, and the locations are consistent with no thread trying to enter or in the critical section, as required in a covering state. Fig. 2.15 illustrates this scenario. □

This line of argument can be extended to show that n-thread deadlock-free mutual exclusion requires n distinct locations. The `Peterson` and `Bakery` locks are thus optimal (within a constant factor). However, the need to allocate n locations per `Lock` makes them impractical.

This proof shows the inherent limitation of read and write operations: Information written by a thread may be overwritten without any other thread ever reading it. We will recall this limitation when we discuss other algorithms.

As discussed in later chapters, modern machine architectures provide specialized instructions that overcome the "overwriting" limitation of read and write instructions, allowing n-thread `Lock` implementations that use only a constant number of memory locations. However, making effective use of these instructions to solve mutual exclusion is far from trivial.

2.10 Chapter notes

Isaac Newton's ideas about the flow of time appear in his famous *Principia* [135]. The "$\rightarrow$" formalism is due to Leslie Lamport [101]. The first three algorithms in this chapter are due to Gary Peterson, who published them in a two-page paper in 1981 [138]. The `Bakery` lock presented here is a simplification of the original Bakery algorithm due to Leslie Lamport [100]. The sequential timestamp algorithm is due to Amos Israeli and Ming Li [85], who invented the notion of a bounded timestamping system. Danny Dolev and Nir Shavit [39] invented the first bounded concurrent

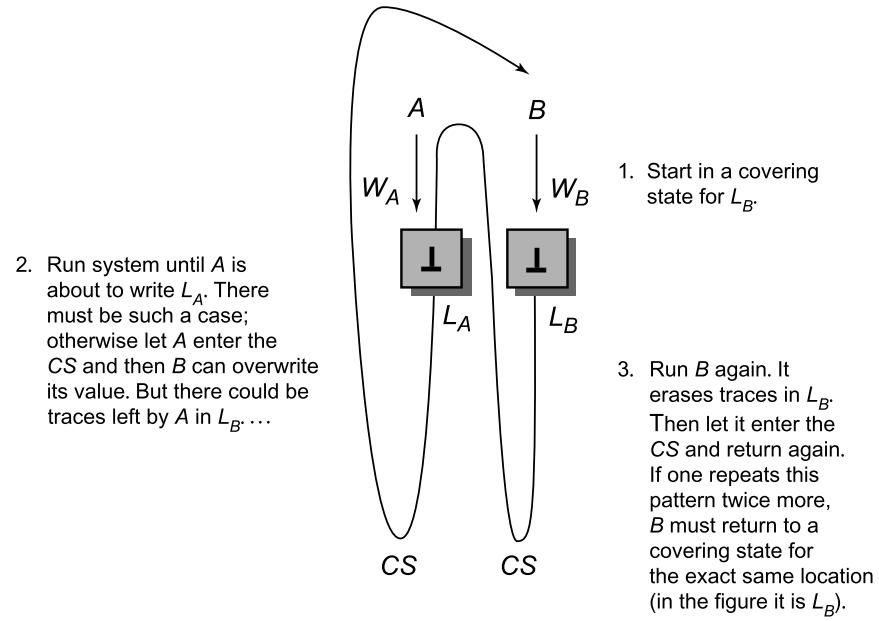

FIGURE 2.15

Reaching a covering state. In the initial covering state for L_B both locations have the empty value ⊥.

timestamping system. Other bounded timestamping schemes include ones by Sibsankar Haldar and Paul Vitányi [56] and Cynthia Dwork and Orli Waarts [42]. The lower bound on the number of lock fields is due to Jim Burns and Nancy Lynch [24]. Their proof technique, called a *covering argument*, has since been widely used to prove lower bounds in distributed computing. Readers interested in further reading can find a historical survey of mutual exclusion algorithms in a classic book by Michel Raynal [147].

2.11 Exercises

Exercise 2.1. A mutual exclusion algorithm provides *r-bounded waiting* if there is a way to define a doorway such that if $D_A^j \rightarrow D_B^k$, then $CS_A^j \rightarrow CS_B^{k+r}$. Does the `Peterson` algorithm provide r-bounded waiting for some value of r?

Exercise 2.2. Why must we define a *doorway* section, rather than defining first-come-first-served in a mutual exclusion algorithm based on the order in which the first instruction in the `lock()` method was executed? Argue your answer in a case-by-case manner based on the nature of the first instruction executed by the `lock()`: a read or a write, to separate locations or the same location.

```
1  class Flaky implements Lock {
2    private int turn;
3    private boolean busy = false;
4    public void lock() {
5      int me = ThreadID.get();
6      do {
7        do {
8          turn = me;
9        } while (busy);
10       busy = true;
11     } while (turn != me);
12   }
13   public void unlock() {
14     busy = false;
15   }
16 }
```

FIGURE 2.16

The Flaky lock used in Exercise 2.3.

```
1  public void unlock() {
2    int i = ThreadID.get();
3    flag[i] = false;
4    int j = 1 - i;
5    while (flag[j] == true) {}
6  }
```

FIGURE 2.17

The revised unlock method for Peterson's algorithm used in Exercise 2.5.

Exercise 2.3. Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 2.16 to achieve n-thread mutual exclusion. For each question, either sketch a proof, or display an execution where it fails.

- Does this protocol satisfy mutual exclusion?
- Is this protocol starvation-free?
- Is this protocol deadlock-free?
- Is this protocol livelock-free?

Exercise 2.4. Show that the `Filter` lock allows some threads to overtake others an arbitrary number of times.

Exercise 2.5. Consider a variant of Peterson's algorithm, where we change the unlock method to be as shown in Fig. 2.17. Does the modified algorithm satisfy deadlock-freedom? What about starvation-freedom? Sketch a proof showing why it satisfies both properties, or display an execution where it fails.

Exercise 2.6. Another 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. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the two-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.) For each of the following properties, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated:

1. mutual exclusion,
2. freedom from deadlock,
3. freedom from starvation.

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?

Exercise 2.7. The ℓ-exclusion problem is a variant of the starvation-free mutual exclusion problem with two changes: Up to 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.

An implementation must satisfy the following conditions:

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

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

Modify the n-thread `Filter` mutual exclusion algorithm to solve ℓ-exclusion.

Exercise 2.8. 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 Fig. 2.18. They claim that if the base `Lock` class provides mutual exclusion and is starvation-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 2.9. Suppose n threads call the `visit()` method of the `Bouncer` class shown in Fig. 2.19. Prove the following:

- 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`. (This is *not* symmetric with the proof for the previous item.)

```
1  class FastPath implements Lock {
2    private Lock lock;
3    private int x, y = -1;
4    public void lock() {
5      int i = ThreadID.get();
6      x = i;                    // I'm here
7      while (y != -1) {}        // is the lock free?
8      y = i;                    // me again?
9      if (x != i)               // Am I still here?
10       lock.lock();            // slow path
11   }
12   public void unlock() {
13     y = -1;
14     lock.unlock();
15   }
16 }
```

FIGURE 2.18

Fast-path mutual exclusion algorithm used in Exercise 2.8.

```
1  class Bouncer {
2    public static final int DOWN = 0;
3    public static final int RIGHT = 1;
4    public static final int STOP = 2;
5    private boolean goRight = false;
6    private int last = -1;
7    int visit() {
8      int i = ThreadID.get();
9      last = i;
10     if (goRight)
11       return RIGHT;
12     goRight = true;
13     if (last == i)
14       return STOP;
15     else
16       return DOWN;
17   }
18 }
```

FIGURE 2.19

The `Bouncer` class implementation for Exercise 2.9.

Exercise 2.10. So far, we have assumed that all n threads have small unique IDs. Here is one way to assign small unique IDs to threads. Arrange `Bouncer` objects in a triangular matrix, where each `Bouncer` is given an ID as shown in Fig. 2.20. Each

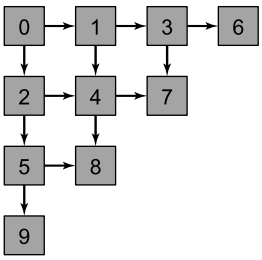

FIGURE 2.20

Array layout for `Bouncer` objects for Exercise 2.10.

thread starts by visiting `Bouncer` 0. 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 do you need in the array if you know in advance the total number n of threads?

Exercise 2.11. Prove, by way of a counterexample, that the sequential timestamp system T^3, starting in a valid state (with no cycles among the labels), does 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 2.12. The sequential timestamp system T^n had a range of $O(3^n)$ different possible label values. Design a sequential timestamp system that requires only $O(n2^n)$ labels. Note that in a timestamp system, one may look at all the labels 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 2.13. Give Java code to implement the `Timestamp` interface of Fig. 2.11 using unbounded labels. Then, show how to replace the pseudocode of the `Bakery` lock of Fig. 2.10 using your `Timestamp` Java code.

Exercise 2.14. We saw earlier the following theorem on the bounds of shared memory for mutual exclusion: Any deadlock-free mutual exclusion algorithm for n threads must use at least n shared registers. For this exercise, we examine a new algorithm that shows that the space lower bound of the above theorem is tight. Specifically, we will show the following:

> **Theorem**: There is a deadlock-free mutual exclusion algorithm for n threads that uses exactly n shared bits.

```
1  class OneBit implements Lock {
2    private boolean[] flag;
3    public OneBit (int n) {
4      flag = new boolean[n]; // all initially false
5    }
6    public void lock() {
7      int i = ThreadID.get();
8      do {
9        flag[i] = true;
10       for (int j = 0; j < i; j++) {
11         if (flag[j] == true) {
12           flag[i] = false;
13           while (flag[j] == true) {} // wait until flag[j] == false
14           break;
15         }
16       }
17     } while (flag[i] == false);
18     for (int j = i+1; j < n; j++) {
19       while (flag[j] == true) {} // wait until flag[j] == false
20     }
21   }
22   public void unlock() {
23     flag[ThreadID.get()] = false;
24   }
25 }
```

FIGURE 2.21

Pseudocode for the `OneBit` algorithm.

To prove this new theorem, we study the `OneBit` algorithm shown in Fig. 2.21. This algorithm, developed independently by J. E. Burns and L. Lamport, uses exactly n bits to achieve mutual exclusion; that is, it uses the minimum possible shared space.

The `OneBit` algorithm works as follows: First, a thread indicates that it wants to acquire the lock by setting its bit to *true*. Then, it loops and reads the bits of all threads with smaller IDs than its own. If all of these bits are *false* (while its own bit is *true*), then the thread exits the loop. Otherwise, the thread sets its bit to *false*, waits until the bit it found to be *true* becomes *false*, and starts all over again. Afterwards, the thread reads the bits of all threads that have IDs greater than its own, and waits until they are all *false*. Once this check passes, the thread can safely enter the critical section.

- Prove that the `OneBit` algorithm satisfies mutual exclusion.
- Prove that the `OneBit` algorithm is deadlock-free.