

Chapter 2

Mutual Exclusion

This chapter covers a number of “classical” mutual exclusion algorithms that work by reading and writing fields of shared objects. Let us be clear: these algorithms are not used in practice. Instead, 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 you will need to understand before you are ready to approach the design of truly practical techniques.

2.1 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 need to articulate complicated conditions involving how multiple time intervals can overlap, or perhaps how they can't. We need a simple but unambiguous language to talk about events and durations in time. Common-sense English is not sufficient, because it 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). Recall that a thread is a state machine, and its state transitions are called *events*. Events are 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, \dots . Threads typically contain loops, so a single program statement can produce many events. It is convenient to denote the j -th occurrence of an event

⁰*This chapter is part of the Manuscript MULTIPROCESSOR SYNCHRONIZATION by Maurice Herlihy and Nir Shavit copyright © 2003, all rights reserved.*

a_i by a_i^j . One event a *precedes* another event b , written $a \rightarrow b$, if a occurs at an earlier time. The “ \rightarrow ” relation is a total order on events.

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). More succinctly, the \rightarrow relation is a partial order on intervals. Intervals that are unrelated by \rightarrow are said to be *concurrent*. By analogy with events, we denote the j -th execution of interval I_A by I_A^j .

2.2 Critical Sections

```
class Counter {
    private int value = 1; // counter starts at one

    public Counter(int c) { // constructor initializes counter
        value = c;
    }

    public int inc() { // increment value & return prior value
        int temp = value; // start of danger zone
        value = temp + 1; // end of danger zone
        return temp;
    }
}
```

Figure 2.1: The Counter class

In an earlier chapter, we discussed the Counter class implementation shown in Figure 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, of course, 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 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 2.2.

The Lock constructor creates a new lock object. A thread calls `acquire` before entering a critical section, and `release` before leaving the critical section. Assume for simplicity that there are n threads labelled 0 through $n - 1$, and that any thread can get its own thread identifier by calling `Thread.myIndex()`. Figure 2.3 shows how to add a lock to the shared counter implementation.

We now formalize the properties a good lock implementation should satisfy. Let CS_A^j be the interval during which A executes the critical section for the j -th

```
public interface Lock {
    public Lock();
    public void acquire(int i); // before entering critical section
    public void release(int i); // before leaving critical section
}
```

Figure 2.2: The Lock Interface

```
public class Counter {
    private long value = 1; // counter starts at one
    private Lock lock;      // to protect critical section
    public Counter(int c) { // constructor initializes counter
        lock = new Lock();
        value = c;
    }
    public long inc() {      // increment value & return prior value
        int i = Thread.myIndex(); // get thread index
        lock.acquire(i);        // enter critical section
        int temp = value;      // in critical section
        value = temp + 1;     // in critical section
        lock.release(i);      // leave critical section
        return temp;
    }
}
```

Figure 2.3: Using a Lock Object

time. Assume, for the sake of the definition, that an execution of a program involves the executing the critical section infinitely often, with other non-critical operations taking place in between. Here are three properties a good mutual exclusion protocol might satisfy.

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 wants in, some thread gets in. If thread A calls `acquire` but never acquires the lock, then other threads must have completed an infinite number of critical sections.

No Lockout Every thread that wants in, eventually gets in. Every call to `acquire` eventually returns.

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

The Mutual Exclusion property is clearly essential. The deadlock-free property is important. It implies that the system never “freezes”. Individual threads may be stuck forever (called *starvation*), but some thread makes progress. Note that a program can still deadlock even if all the locks it uses are individually deadlock free: for example, A and B may try to acquire two locks in different orders.

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 a prerequisite for understanding whether it is a realistic threat.

The lockout-free property is also weak in the sense that makes no guarantees 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.

2.3 Two-Thread Solutions

We begin with two inadequate but interesting algorithms.

2.3.1 The Lock1 Class

Lock implementation 1 is depicted in Figure 2.4. Our two-thread lock implementations follow the following conventions: the threads have indexes 0 and 1, the calling thread has index i , and the other thread has index $j = 1 - i$.

We use $write_A(x = v)$ to denote the event in which A assigns value v to variable or field x , and $read_A(v == x)$ to denote the event in which A reads v from variable or field x . Sometimes we omit v when the value is unimportant. For example, in Figure 2.4 $write_1(flag[i] = true)$ is the first line of the `acquire` method.

```

class Lock1 implements Lock {

    private bool flag[2];

    public void acquire(int i) {
        int j = i-1;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void release(int i) {
        flag[i] = false;
    }
}

```

Figure 2.4: Lock Implementation 1

Lemma 2.3.1 *The Lock1 implementation 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 `acquire` method before entering its k -th (j -th) critical section.

Inspecting the code, we see that

$$write_A(flag[A] = true) \rightarrow read_A(flag[B] == false) \rightarrow CS_A \quad (2.1)$$

$$write_B(flag[B] = true) \rightarrow read_B(flag[A] == false) \rightarrow CS_B \quad (2.2)$$

$$read_A(flag[B] == false) \rightarrow write_B(flag[B] = true) \quad (2.3)$$

Note that once $flag[B]$ is set to *true* it remains *true*. It follows that Equation 2.3 holds, since otherwise thread A could not have read $flag[B]$ as *false*.

$$\begin{aligned}
 &write_A(flag[A] = true) \rightarrow \hspace{15em} (2.4) \\
 &read_A(flag[B] == false) \rightarrow write_B(flag[B] = true) \rightarrow read_B(flag[A] == false)
 \end{aligned}$$

Equation 2.4 follows from Equations 2.1, 2.3 and 2.2, and from the transitivity of precedence. It follows that $write_A(flag[A] = true) \rightarrow read_B(flag[A] == false)$ without an intervening $write_A(flag[A] = false)$, a contradiction. ■

Why is the `Lock1` class inadequate? It deadlocks if thread executions are interleaved. If both $write_A(flag[A] = true)$ and $write_B(flag[B] = true)$ events occur before $read_A(flag[B])$ and $read_B(flag[A])$ events, then both threads wait forever. Why is the `Lock1` class interesting? If one thread runs before the other, no deadlock occurs, and all is well.

2.3.2 The Lock2 Class

Figure 2.5 shows an alternative lock implementation, the `Lock2` class.

```
class Lock2 implements Lock {
    private int victim;

    public void acquire(int i) {
        victim = i; // ultimate sacrifice...you go first
        while (victim == i) {}
    }

    public void release(int i) {}
}
```

Figure 2.5: Lock2 Implementation

Lemma 2.3.2 *The Lock2 implementation 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 as before each thread's last execution of the `acquire` method before entering its k -th (j -th) critical section.

Inspecting the code, we see that

$$write_A(victim = A) \rightarrow read_A(victim == B) \rightarrow CS_A \quad (2.5)$$

$$write_B(victim = B) \rightarrow read_B(victim == A) \rightarrow CS_B \quad (2.6)$$

Thread B must assign B to the `victim` field between events $write_A(victim = A)$ and $read_A(victim = B)$ (see Equation 2.5). Since this assignment is the last, we have

$$write_A(victim = A) \rightarrow write_B(victim = B) \rightarrow read_A(victim == B) \quad (2.7)$$

Once the `victim` field is set to B , it does not change, so any subsequent read will return B , contradicting Equation 2.6. ■

Why is the `Lock2` class inadequate? It deadlocks if one thread runs completely before the other. Why is the `Lock2` class interesting? If the threads run concurrently, the `acquire` method succeeds. The `Lock1` and `Lock2` classes complement one another: each succeeds under conditions that cause the other to deadlock.

2.3.3 The Peterson Lock

We now combine the `Lock1` and `Lock2` implementations to construct a **Peterson** implementation of `Lock` that is lockout-free. This algorithm is arguably the most

```

class Peterson implements Lock {

    private bool flag[2];
    private int victim;

    public void acquire(int i) {
        int j    = 1-i;
        flag[i] = true; // I'm interested
        victim   = i;   // you go first
        while (flag[j] && victim == i) {}; // wait
    }

    public void release(int i) {
        flag[i] = false; // I'm not interested
    }
}

```

Figure 2.6: Peterson Lock Implementation.

succinct and elegant two-thread mutual algorithm. It is known as “Peterson’s Algorithm”, after its inventor.

We now sketch a correctness proof.

Lemma 2.3.3 *The Peterson lock satisfies mutual exclusion.*

Proof: Suppose not. As before, consider the last executions of the `acquire` method. Inspecting the code, we see that

$$write_A(flag[A] = true) \rightarrow \quad (2.8)$$

$$write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim) \rightarrow CS_A$$

$$write_B(flag[B] = true) \rightarrow \quad (2.9)$$

$$write_B(victim = B) \rightarrow read_B(flag[A]) \rightarrow read_B(victim) \rightarrow CS_B$$

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

$$write_B(victim = B) \rightarrow write_A(victim = A) \quad (2.10)$$

Equation 2.10 implies that A read `victim` to be A in Equation 2.8. Since A nevertheless entered its critical section, it must have read `flag[B]` to be *false*, so we have:

$$write_A(victim = A) \rightarrow read_A(flag[B] == false) \quad (2.11)$$

Equations 2.9, 2.10, and 2.11, and transitivity of \rightarrow imply Equation 2.12.

$$\begin{aligned} & \text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{write}_B(\text{victim} = B) \rightarrow \\ & \text{write}_A(\text{victim} = A) \rightarrow \text{read}_A(\text{flag}[B] == \text{false}) \end{aligned} \quad (2.12)$$

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

Lemma 2.3.4 *The Peterson lock implementation is lockout-free.*

Proof: Suppose not. Suppose (without loss of generality) that A runs forever in the `acquire` method. 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 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 `acquire` method, a contradiction.

So it must be that B is also be stuck in its call to the `acquire` method, 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 class is deadlock-free.*

2.4 N -Thread Solutions

We now consider two mutual exclusion protocols that work for N threads, where N is greater than 2. The first solution, the *Filter* algorithm, is a direct generalization of Peterson’s algorithm to multiple threads. The second solution, the *Bakery* algorithm, is perhaps the simplest and most elegant known solution to this problem.

2.4.1 Filter Algorithm

The `Filter` algorithm, shown in Figure 2.7, 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 among all that want to enter level ℓ succeeds in doing so.
- If more than one thread wants to enter level ℓ , then at least one is blocked from doing so.

The **Peterson** algorithm used a two-element boolean **flag** array to indicate whether a thread is interested in entering the critical section. The **Filter** algorithm generalizes this notion with an N -element integer **level** array, where the value of **level**[i] indicates the latest level that thread i is interested in entering.

The *filter algorithm* forces each thread to pass through $N - 1$ levels of “exclusion” to enter its critical section. Each level L has a distinct *victim*[L] field, which is used to “filter out” one thread, preventing it from reaching the next level. This array is the natural generalization of the **victim** field in the two-thread algorithm.

```
class Filter implements Lock {

    private bool level[N];
    private int  victim[N-1];

    public void acquire(int i) {
        for (int j = 1; j < N; j++) {
            level[i] = j;
            victim[j] = i;
            while ((exists k != i) level[k] >= j) && victim[j] == i
                ); // busy wait
        }

    public void release(int i) {
        level[i] = 0;
    }
}
```

Figure 2.7: Filter Lock Implementation

We say that a thread A is at *level* 0 if $level[A] = 0$, and at *level* j for $j > 0$, if it last completed the busy waiting loop with $level[A] \geq j$. This definition implies that a thread at level j is also at level $j - 1$, and so on.

Lemma 2.4.1 *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 :

$$write_B(victim[j]) \rightarrow write_A(victim[j])$$

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

$$write_B(level[B] = j) \rightarrow write_B(victim[j]) \rightarrow write_A(victim[j])$$

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

$$write_B(level[B] = j) \rightarrow write_B(victim[j]) \rightarrow write_A(victim[j]) \rightarrow read_A(level[B])$$

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 $n - 1$.

Corollary 2.4.2 *The Filter algorithm satisfies mutual exclusion.*

Theorem 2.4.3 *The Filter 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 2.4.4 *The Filter implementation is deadlock free.*

2.5 Fairness

The lockout-free property guarantees that every thread that calls `acquire` will eventually enter the critical section, but it makes no guarantees about how long it will take. Ideally, and very informally, if A calls `acquire` before B , then A should enter the critical section before B . Such a guarantee is impossible to provide with the tools at hand, and is stronger than we really need. Instead, we split the `acquire` method into two intervals:

```

class Bakery implements Lock {
    boolean flag[n];
    int label[n];

    public void acquire(int i) {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n])+1;
        while (exists k != i such that
            flag[k] && (label[i],i) > (label[k],k));
    }

    public void release(int i) {
        flag[i] = false;
    }
}

```

Figure 2.8: Algorithm 5: The Bakery Algorithm

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

As usual, we use superscripts to indicate repetition.

Here is how we define fairness.

Definition *r-bounded waiting property*: For threads A and B and integers j and k ,

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

In English: if thread A finishes its doorway before thread B starts its doorway, then A can be “overtaken” at most r times by B . The strong form of fairness known as *first-come-first-served* is equivalent to *0-bounded waiting*.

2.6 The Bakery Algorithm

The Bakery algorithm shown in Figure 2.8. It satisfies the *first-come-first-served* property by assigning each thread a “number” in the doorway interval. In the waiting interval, the thread waits until no thread with an earlier number is trying to enter the critical section.

The $flag[A]$ field is a boolean variable written only by A to indicate whether A wants to enter the critical section. The $label[A]$ field is an integer value that determines A ’s order among the threads trying to reach the critical section. Comparisons are done on pairs consisting of a label and a thread index. Pairs are ordered lexicographically: pair $(a, b) > (c, d)$ if $a > b$, or if $a = b$ and $b > d$.

It is easy to see that a thread's labels are strictly increasing. Threads may have the same label, but thread indexes break any ties when comparing pairs.

Lemma 2.6.1 *The Bakery algorithm is deadlock free.*

Proof: Some waiting thread A has the unique least $(label[A], A)$ pair, and that thread can return from the `acquire` method. ■

Lemma 2.6.2 *The Bakery algorithm is first-come-first-served.*

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

$$D_A \rightarrow D_B$$

then A 's label is smaller since

$$write_A(label[A]) \rightarrow read_B(label[A]) \rightarrow write_B(label[B]) \rightarrow read_B(flag[A]),$$

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

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

Lemma 2.6.3 *The Bakery 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 respective sequences of acquiring a new label by choosing one greater than all those read. Suppose $(label[A], A) < (label[B], B)$. When B entered, it must have seen that $flag[A]$ was *false* or that $label[A] > label[B]$.

But labels are strictly increasing, so B must have seen that $flag[A]$ was *false*. It follows that

$$labeling_B \rightarrow read_B(flag[A]) \rightarrow write_A(flag[A]) \rightarrow labeling_A$$

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

2.7 Bounded Timestamps

Notice that labels 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 (in the Bakery algorithm even mutual exclusion will be violated) no longer holds.

Later in the book we will see a number of constructions where counters are used to order threads, or even to produce unique identifiers. How important is the overflow problem in the real world? 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 real-world overflow problem, even if the consequences were not as dire as predicted. On 18 January 2038, the Unix

```

public interface Timestamp {
    boolean compare(Timestamp);
}

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

```

Figure 2.9: Timestamping System

`time_t` data structure will overflow when the number of seconds since 1 January 1970 exceeds 2^{16} . 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 us focus on the role of the `label` values in the Bakery algorithm. 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 Algorithm 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 2.9.

Can we construct such an object? Ideally, since we are implementing mutual exclusion (the timestamping system will be for example part of the doorway of the Bakery Algorithm) any such algorithm should be wait-free. It turns out that it is possible to implement such a wait-free *concurrent timestamping* system. Unfortunately, the full construction is long and rather technical, so instead we will focus on a simpler problem, a *sequential timestamping* system, in which we assume that a thread can scan and label in a single atomic step. The principles are essentially the same as in the concurrent case, but the details are much simpler.

Think of the set of timestamps as nodes of a directed graph (called a *precedence graph*). There is an edge from node *a* to node *b* if *a* is a later timestamp than *b*. The timestamp order is *irreflexive*: there is no edge from any node *a* to itself. It 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*: an edge from *a* to *b* and from *b* to *c* does not necessarily mean there is an edge from *a* to *c*.

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

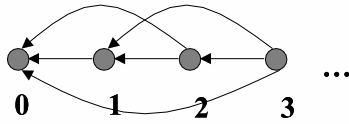


Figure 2.10: Precedence graph for unbounded timestamping system

Pragmatically, we can implement such a system as an array of single-writer multi-reader registers, 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.

The precedence graph for the unbounded counter used in the Bakery array appears in Figure 2.10. Not surprisingly, the graph is infinite: there is one node for each natural number, with a directed edge from a to b whenever $a > b$.

Let us use $a \succ b$ to mean that there is an edge from a to b . Consider the precedence graph T^2 graph shown in Figure 2.11. This graph has three nodes, labelled 0, 1, and 2, where $0 \succ 1$, $1 \succ 2$, and $2 \succ 0$. If there are only two threads, then we can use this graph to define a bounded (sequential) timestamping system. The two threads occupy adjacent nodes, with the direction of the edge indicating their order. Suppose A has 0, and B has 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* in a directed graph is a set of nodes n_0, n_1, \dots, 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 word “cycle” comes from the same Greek root as “circle”.

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 $k > 2$, $T^k = T^2 \circ T^{k-1}$.

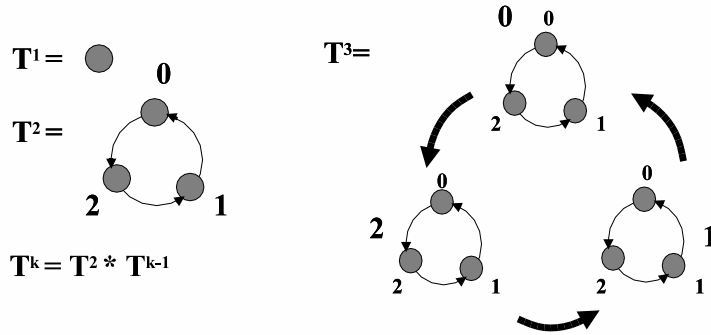


Figure 2.11: Precedence graph for bounded timestamping system

For example, the graph T^3 is illustrated in Figure 2.11.

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, \dots , 22, where the high-order digit indicates one of the three subgraphs, and the low-order digit indicates one node within that subgraph. The generalization to multiple levels should be obvious.

The key to understanding the n -thread labelling 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.

2.8 Lower Bounds on Number of memory Fields

The Bakery algorithm is succinct, elegant, and fair. So why isn't it considered practical? The principal drawback is the need to read n distinct fields, where n is the maximum number of concurrent threads. The number n may be very large, fluctuating, or even unknown. Moreover, threads must be assigned unique indexes between 0 and $n - 1$, which is awkward if threads can be created and destroyed dynamically.

Is there an even cleverer algorithm that avoids these problems? There do exist “fast-path” mutual exclusion algorithms where the number of fields read or written is proportional to the number of threads simultaneously trying to acquire the locks. Nevertheless, one can prove that any deadlock-free mutual exclusion algorithm still requires accessing at least n distinct fields in the worst case.

Theorem 2.8.1 *Any algorithm that solves deadlock free mutual exclusion must use n distinct memory locations.*

Lets understand how one would prove this fact for three threads, that is, that three threads require at least three distinct memory fields to solve mutual exclusion with no deadlock in the worst case.

Theorem 2.8.2 *There is no algorithm that solves deadlock free mutual exclusion for three threads using less than three distinct memory locations.*

Proof: Assume by way of contradiction that there is such an algorithm for processors A , B , and C , and it uses two variables. It should be clear that each thread must write to some variable, otherwise we can first run that thread into the critical section. Then run the other two threads and since the first thread did not write there will be no trace in the two shared variables that he is in the critical section and one of the other threads will also enter the critical section, violating mutual exclusion. It follows that if the shared variables are single-writer variables as in the Bakery algorithm, than it is immediate that three separate variables are needed.

It remains to be shown that the same holds for multi-writer variables as such as `victim` in Peterson's algorithm. To do so, lets define the notion of a *covering state*.

Definition A covering state is one in which there is at least one thread thread poised to write to each shared variable, and the state of the shared variables is consistent with all threads not being in the critical section or trying to enter the critical section.

If we can bring threads A and B to a covering state where they respectively cover the two variables R_A and R_B , then we can run thread C and it will have to enter the critical section. Though it will have to write into R_A or R_B or both, if we run both A and B since they were in a covering state their first step will be to overwrite any information left by C in the shared variables and so they will be running in a deadlock free mutual exclusion algorithm and one of them will enter and join C in the critical section, a violation of mutual exclusion. This scenario is depicted in Figure 2.12.

It thus remains to be shown how to manoeuver A and B into a covering state. We will do so as follows. Consider an execution in which thread B runs through the critical section three times. We have already shown that each time it must write some variable, and lets look at the first variable it is about to write in each round through the critical section. Since there are only two variables, B must, during this execution, be twice poised to perform its first write to the same variable. Lets call that R_B .

Now, run B till its poised to write R_B for the first time. Then run A until it is about to write to variable R_A for the first time. A must be on its way to enter the critical section since B has not written. It must write R_A at some point before entering the critical section since otherwise, if it only writes to R_B , we can run B , obliterate any trace of A in R_B , and then have B enter the critical section together with R_A , violating mutual exclusion.

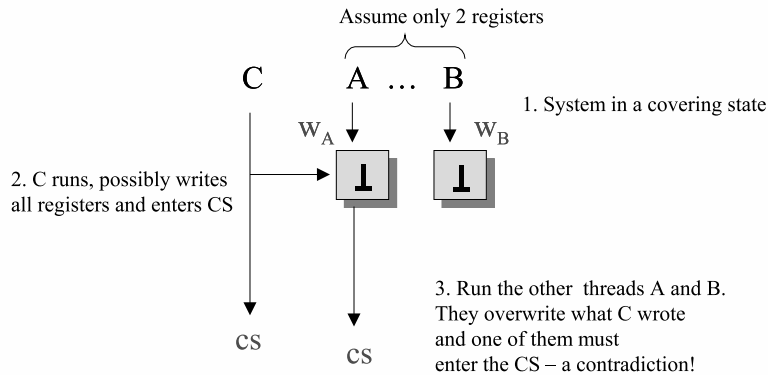


Figure 2.12: Contradiction using covering state for two variables

Now, on its way to writing R_A , A could have left traces in R_B . Here we use the fact that we can run B , and since it was poised to write it will obliterate any traces of A and enter the critical section. If we let B enter the critical section twice more, it will as we have shown earlier have to reach a position where it is poised to first write R_B . This places A poised to write to R_A and B poised to write to R_B , and the variables are consistent with no thread trying or in the critical section, as required in a covering state. This scenario is depicted in Figure 2.13. ■

In later chapters, we will see that modern machine architectures provide specialized instructions for synchronization problems like mutual exclusion. We will also see that making effective use of these instructions is far from trivial.

2.9 Granularity of Mutual Exclusion

We round up this chapter with a discussion of how mutual exclusion can be used in practice. Figure 2.14 shows a standard Java implementation of a shared FIFO queue. To understand this code, you must be aware of how Java handles synchronization. Each Java object has an implicit lock field and an implicit con-

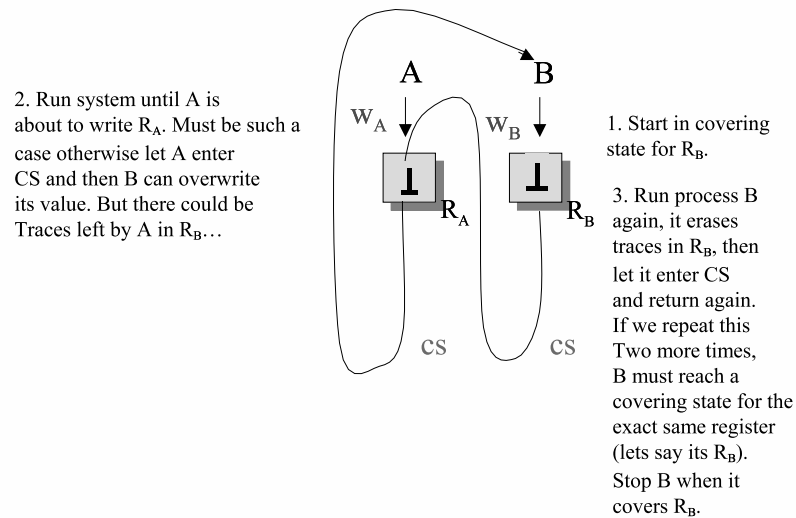


Figure 2.13: Reaching a covering state

dition 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 method calls lock the entire queue, and cannot proceed in parallel. 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 2.15 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` items,
- `tail` is the index in the `items` array at which the next enqueued item will be stored `head` is the index in the `items` array from which the next dequeued item will be removed

```

class Queue {
    int head = 0;           // next item to dequeue
    int tail = 0;          // next empty slot
    Item[QSIZE] items;

    public synchronized void enq(Item x) {
        while (this.tail - this.head == QSIZE) {
            try {
                this.wait();           // wait until not full
            } catch (InterruptedException e) {}; // ignore exceptions
        }
        this.items[this.tail++ % QSIZE] = x;
        this.notifyAll();
    }

    public Item deq() {
        while (this.tail - this.head == 0) {
            try {
                this.wait();           // wait until non-empty
            } catch (InterruptedException e) {}; // ignore exceptions
        }
        Item x = this.items[this.head++];
        this.notifyAll();
        return x;
    }
}

```

Figure 2.14: Lock-based FIFO Queue

```

class Queue {

    int head = 0;           // next item to dequeue
    int tail = 0;          // next empty slot
    Item[QSIZE] items;

    public void enq(Item x) {
        while (this.tail - this.head == QSIZE); // busy-wait
        this.items[this.tail % QSIZE] = x;
        this.tail++;
    }

    public Item deq() {
        while (this.tail == this.head); // busy-wait
        Item x = this.items[this.head % QSIZE];
        this.head++;
        return x;
    }
}

```

Figure 2.15: FIFO Queue without Mutual Exclusion

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 item 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 native 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 multiprocessor to its knees.

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 an n -way symmetric 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 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 on an n -way multiprocessor. Let c be the fraction of the program that can be executed in parallel, 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 -way multiprocessor:

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

For example, if a program spends 20% of its time in critical sections, and is deployed on a 10-way multiprocessor, 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.

2.10 Chapter Notes

The first three algorithms in this chapter are due to Gary Peterson. The Bakery Algorithm we present is a simplification of the original Bakery Algorithm due to Leslie Lamport. The lower bound on the number of memory locations needed to solve mutual exclusion is due to Burns and Lynch. The sequential one we describe here is due to Israeli and Li, who are the originators of the concept of a bounded timestamp system. The first bounded concurrent timestamping system was provided by Dolev and Shavit.

2.11 Exercises

1. Programmers at the Flaky Computer Corporation designed the following protocol for n -process mutual exclusion with deadlock-freedom.

```
class Flaky implements Lock{

    private int turn;
    private bool busy = false; // initially to false

    void acquire(int i){        // code for thread i
        do {
            do {                // loop until this.busy is false
                this.turn = i;
            } while (this.busy);
            this.busy = true;
        }
    }
}
```

```

    } while (this.turn != i);
}

void release(int i){
    this.busy = false;
}
}

```

Does this protocol satisfy mutual exclusion? Either sketch a proof, or display an execution where it fails. Is this a safety, liveness or fairness property?

Does this protocol satisfy no-lockout? Either sketch a proof, or display an execution where it fails. Is this a safety, liveness or fairness property?

Does this protocol satisfy no-deadlock? Either sketch a proof, or display an execution where it fails. Is this a safety, liveness or fairness property?

Does this protocol satisfy no-starvation (i.e., every process that wants to get in the critical section, gets in eventually)? Either sketch a proof, or display an execution where it fails. Is this a safety, liveness or fairness property?

What are the differences between satisfying no-lockout, satisfying no-deadlock and satisfying no-starvation? Is any of these concepts stronger than another?

2. Does the Filter algorithm provide fairness? Is there an r such that the filter algorithm provides r -bounded waiting?
3. Another way to generalize the two-thread Peterson mutual exclusion algorithm 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 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.

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

Either sketch a proof that this tree-lock satisfies no-lockout, or given 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?

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 any thread that wants to enter the critical section will eventually succeed (even if some threads in the critical section have halted).

Modify the Peterson n -process mutual exclusion algorithm to make it into an ℓ -exclusion algorithm.

5. In chapter two we discussed time-stamp systems.
 - (a) Prove, by way of a counter example, that the sequential time-stamp system T^3 in Chapter 2, 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 counter example should thus bring about a situation where in some state of the execution three labels are not totally ordered.
 - (b) The sequential time-stamp system in Chapter 2 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.

2.12 Bibliography

- L. Lamport, *A New Solution of Dijkstra's Concurrent Programming Problem*, Communications of the ACM, 17 (8), 453-455, 1974.
- G. L. Peterson, *Myths About the Mutual Exclusion Problem* Information Processing Letters, 12(3), pages 115-116, 1981.
- A. Israeli and M. Li. Bounded timestamps. *Distributed Computing*, 6(4):205-209, 1993.

- T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*, Prentice-Hall, Inc. pp 31-32, 38-39, 1992.
- D. Dolev, and N. Shavit, *Bounded Concurrent Time-Stamping*, SIAM Journal on Computing, Vol. 26, No. 2, pp. 418-455, 1997.
- J. E. Burns and N. A. Lynch. *Bounds on shared memory for mutual exclusion*, Information and Computation, 107(2):171-184, December 1993.