Exercise 9. Define *r-bounded waiting* for a given mutual exclusion algorithm to mean that if $D_A^j \rightarrow D_B^k$ then $CS_A^j \rightarrow CS_B^{k+r}$. Is there a way to define a doorway for the Peterson algorithm such that it provides r-bounded waiting for some value of r?

Solution The Peterson lock divides naturally into doorway and waiting sections: the **while** loop is the waiting section, and the rest is the doorway.

Supose A completes doorway section D_A^j, then B starts a doorway D_B^k such that $D_A^j \rightarrow D_B^k$. At the time B's doorway starts, victim could be either A or B. If victim is B, then any lock() call by B that starts after A's doorway will be blocked until A resets victim:

$$D_A^j \rightarrow D_B^k \Rightarrow CS_A^j \rightarrow CS_B^k$$

If victim is A, then B will acquire and release the lock, but B's next doorway will set victim is B:

$$D_A^j \rightarrow D_B^k \Rightarrow CS_A^j \rightarrow CS_B^{k+1}$$

In this way, the Peterson algorithm is 1-bounded waiting, that is, FCFS.

Exercise 10. Why do we need to define a *doorway* section, and why cannot we define FCFS 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.

Solution Any mutual exclusion algorithm consists of computation and waiting. By splitting the code into a doorway section followed by a waiting section, we move all the computation to the front (instead of interleaving computation and waiting). This imposes a clean structure on the algorithm, and makes it easier to recognize and eliminate unneeded blocking (that is, blocking during the doorway section).

The problem with defining FCFS in terms of the first step taken by the doorway is that usually it's impossible to tell who took the first step. Consider the following two-thread cases:

- The first step is a read. If the two threads read one right after the other, it is impossible to tell which went first.
- The first step is a write, and two threads write to different locations. If the two threads write one right after the other, it is impossible to tell which went first.
- The first step is a write, and both threads write to the same location. In one scenario, B writes, A writes, so FCFS requires B to enter the critical section first. In another scenario, A alone writes, so FCFS requires A to enter the critical section first. Problem is, A cannot distinguish between these two sitations, since its write obliterated any evidence that B may have written first.

Exercise 11. Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 1 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?

Solution The Flaky algorithm satisfies Mutual exclusion. Assume that threads A and B are in the critical section at the same time. Suppose A went past Line 11 before B. A must have seen that turn was equal to A. The only way B could have returned from the lock() method is if it saw turn equal to B. The only way turn can become equal to B after being equal to A is if B sets it at Line 8 after A has entered the critical section. But since A set busy to *true* before it entered the critical section, B could not have left the inner loop.

The flaky algorithm can deadlock. Starting with the critical section empty (busy == *false*):

- A calls lock();
- A sets turn to A;
- A sets turn to A, and busy to *true*;
- B calls lock();
- B sets turn to B.

B is now stuck in the inner loop until A releases the lock. A, however, will repeat the outer loop since turn is not equal to A. A will then be stuck in the inner loop until B releases the lock. Any additional thread that tries to acquire the lock will also get stuck in the inner loop.

This algorithm is not starvation-free, because it deadlocks.

Exercise 12. Show that the Filter lock allows some threads to overtake others an arbitrary number of times.

Solution Given three threads A, B, and C, C can overtake A an infinite number of times.

1. C acquires the lock.
2. A calls lock(), becoming the victim at level 1.
3. B calls lock(), becoming the victim at level 1.
4. C releases the lock, calls lock() again, and becomes the victim at level 1.

5. B acquires and releases the lock.

6. B calls lock(), becoming the victim at level 1.

7. C acquires the lock.

This proceeds until A wakes up and moves on.

Exercise 13. Another way to generalize the two-thread Peterson lock is to arrange a number of 2-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 2-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 property, 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?

Solution Fig. 2 shows the TreeLock constructor. First, we make sure that the number of threads is a power of two. Then it creates the tree of locks. We calculate the tree minus one), then we intialize the locks in the array. We assign a different mask to each level in the tree, so we can derive the index from the thread ID.

The lock() method first identifies that thread's leaf. Then it loops from the leaf to the root, and locks the nodes. When a node is locked, we remember the thread id so we can unlock it later.
The unlock() method unlock the nodes from the root to the leaf.

The actual lock implementation is just the Peterson lock, except that we use the mask to identify which thread, by applying a bit-wise OR to the thread ID.

1. *Mutual Exclusion:* The tree-lock guarantees mutual exclusion. Since each of the tree nodes is a Peterson lock, as long as none of these locks are accessed by more than two threads concurrently, they provide mutual exclusion - that is, only one thread can acquire each lock and proceed to the parent node (or enter the critical section in the case of the root node). We can therefore prove that the tree-lock guarantees mutual exclusion by

induction on the depth of the tree: For tree of depth 1, the tree is just a Peterson lock and so provides mutual exclusion. Assume trees of depth $d-1$ provide mutual exclusion. Consider a tree of depth d. The root node of that tree can be accessed only by threads that acquire the left or the right subtrees rooted at it. Since each of these subtrees provides mutual exclusion, at most one thread can acquire each of them, and therefore at most two access the root node. Therefore at most one thread acquires the root node and enters the critical section.

2. *Deadlock and Starvation freedom:* The tree-lock guarantees freedom from deadlock and starvation. If any thread is blocked forever by the tree-lock, then it must be blocked forever by one of the Peterson locks in the tree. But each of these locks is starvation-free, a contradiction.

3. *Bounded overtaking:* We cannot specify an upper bound on overtaking[1] because while a thread is delayed, other threads can acquire any lock not yet visited by that thread an arbitrary number of items. Consider, for example, the case of 4 threads with a two-layer tree (two leaves), where one thread acquires the left leaf and is then delayed for some arbitrary finite period of time. During this period, another thread can acquire the right leaf and then the root node an arbitrary number of times. Therefore, even though the first thread has already passed its doorway, another thread that enters its doorway after that acquires the tree-lock before the first thread does an arbitrary number of times, and that can happen an arbitrary amount of times.

Exercise 14. The ℓ-exclusion problem is a variant of the starvation-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.

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, then 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-process Filter mutual exclusion algorithm to turn it into an ℓ-exclusion algorithm.

[1] We assume that the doorway for a thread is defined to be the writes to the flag and victim variables in the appropriate leaf node. A thread t_1 is overtaken by thread t_2 if thread t_1's doorway precedes t_2's doorway, but t_2 acquires the tree-lock before t_1 does.

Solution To adapt the filter algorithm to do ℓ-exclusion we must modify two parts. First, we allow ℓ threads into the critical section by decreasing the number of fillter levels: if exactly one thread is the victim at each level, having $n-\ell$ levels allows ℓ threads into the critical region. The second property is more subtle. In the filter algorithm, if one thread is in the critical region the algorithm's test at every level blocks other threads at each and every level. The problem with this code in the ℓ-exclusion case is that we must let threads in, in fact, even if there are $\ell-1$ threads in the critical section, we must allow one more thread to enter. To ensure that ℓ threads can always enter the critical section, the test by any thread i at any given level k is modified to ensure that at lease $n-\ell$ threads are at levels before the current thread. If there are $n-\ell$ before it, then there can be at most ℓ (including itself) either in the critical or at a level greater than k in the trying region. The code for the solution appears in Figure 1.

Exercise 15. 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. 6. 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.

Solution The wrapper fails to provide mutual exclusion:

1. Thread 0 sets x to 0, observes y is -1.

2. Thread 1 sets x to 1, observes y is -1, sets y to 1, observes x is 1, enters critical section via the fast path.

3. Thread 0 sets y to 0, observes x is 1, calls lock, enters critical section.

Exercise 16. Suppose n threads call the visit () method of the Bouncer class shown in Fig. 7. Prove that—

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

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

Solution Here is how to show that at most one thread gets the value STOP. The key point is that last can be the ID of at most one of the threads that read goRight+ to be *false* in the **if** statement. We can conclude this because we know

$$last.write(i) \rightarrow goRight.read(false) \rightarrow goRight.write(true) \rightarrow last.read(i) \rightarrow return(\mathsf{STOP})$$

For two threads A and B to have returned STOP, they would have had to have read their ids in last. Without loss of generality, say B read last after A read its value there.

$$A : last.read(A) \rightarrow B : last.read(B)$$

It follows that B wrote its value into last after A had written and read its value there (since A must have written its value to have read it):

$$A : last.write(A) \rightarrow A : last.read(A) \rightarrow B : last.write(B) \rightarrow B : last.read(B)$$

But we know from above that

$$A : last.write(A) \rightarrow A : goRight.write(true) \rightarrow A : last.read(A)$$

and

$$B : last.write(B) \rightarrow B : goRight.read(false) \rightarrow B : last.read(B)$$

Linking these together gives

$$A : last.write(A) \rightarrow A : goRight.write(true) \rightarrow A : last.read(A)$$

$$\rightarrow B : last.write(B) \rightarrow B : goRight.read(false) \rightarrow B : last.read(B)$$

Since it is impossible for any thread to write *false* to goRight, there is no way that the value of goRight could have turned *false* once A wrote it, contradicting B's read of *false*.

Therefore, more than one thread cannot get the value STOP. Here are the steps in the proof.

Claim At most $n-1$ threads get the value DOWN.

Proof For any thread to get the value DOWN, we see that they must have read goRight to be *false* (so they couldn't return RIGHT). Assume n threads were able to do this. Then to have returned DOWN, all threads would have had to have read last to be unequal to their value. But since all n threads have already completed the line where they set last to their id, there are no other threads left to change last's value. We know that some thread must have been the last one to set last to its id, so it must read this and return STOP, contradicting our assumptionthat n threads were able to return DOWN.

Claim At most $n-1$ threads get the value RIGHT.

Proof First off, we see the following holds for returning RIGHT:

$$goRight.read(true) \rightarrow return(\mathsf{RIGHT})$$

We also note that goRight is initialized to false and that there is no write to goRight before it is read by the first thread:

$$goRight.write(false) \rightarrow goRight.read(false) \rightarrow goRight.write(true)$$

So the first thread to check the value of goRight must find it to be false, making it skip the line to return RIGHT. Therefore, all n threads cannot get the value RIGHT since the first one cannot.

Exercise 17. So far, we have assumed that all n threads have unique, small indexes. Here is one way to assign unique small indexes to threads. Arrange Bouncer objects in a triangular matrix, where each Bouncer is given an id as shown in Fig. 8. Each thread starts by visiting Bouncer zero. If it gets STOP, it stops. If it gets RIGHT, it visits 1, and if it gets DOWN, it visits 2. In general, if a thread gets STOP, it stops. If it gets RIGHT, it visits the next Bouncer on that row, and if it gets DOWN, it visits the next Bouncer in that column. Each thread takes the id of the Bouncer object where it stops.

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

Solution We can use the results of the last problem to show that each thread eventually stops at a Bouncer object. Since at most $n-1$ threads get DOWN and at most $n-1$ threads get RIGHT, the problem simplifies recursively at each step of the way, even if no thread stops at that Bouncer. Once the problem simplifies down to only a thread at a Bouncer, that thread must stop since it fails the conditional to go right and then realizes it is the last (and only) thread. We can also conclude this since we know $n-1=0$ threads would get RIGHT and LEFT, so the thread must get STOP.

The number of Bouncer objects needed is $(n-1)*(n-1)/2$. We can do a proof by induction that at each level k (where the zero Bouncer is at the 0 level, the 1 and 2 Bouncers are at the first level, etc.) The maximum number of threads at a Bouncer is $n-k$. Then, at the n^{th} level there should be no more threads, so we can stop creating Bouncers at the $n-1^{st}$ level. This is a tree of height $n-1$, whose number of Bouncers we can calculate by adding up the Bouncers at each level and using the formula for computing an arithmetic series:

$$1+2+3+...+(n-2)+(n-1) = (n-1)*\frac{1+(n-1)}{2} = \frac{n*(n-1)}{2}$$

Exercise 18. Prove, by way of a counterexample, that the sequential time-stamp system T^3, started in a valid state (with no cycles among the labels), 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.

Solution Assume each thread can read the graph atomically, and move its token atomically, but the read and write together are not atomic.

Start the system with A on 01, B on 02, and C on 20.

1. A reads the graph, and prepares to move to 21.
2. B reads the graph, and moves to 21.
3. C reads the graph, and moves to 22.
4. B reads the graph, and moves to 20.
5. A moves to 21.

At this point, the tokens are not totally ordered.

Exercise 19. The sequential time-stamp system T^3 had a range of 3^n different possible label values. Design a sequential time-stamp system that requires only $n2^n$ labels. Note that in a time-stamp system, one may look at all the labels 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.

Solution If there are $O(n2^n)$ labels, we can represent them using $n+\log n$ bits.

Let each label (i,s) consist of the $\log n$ bits of the thread id i and an additional string of n bits, where n is the number of threads. Let $s[k]$ denote the kth bit of the string s. The ordering between two labels (i,s) and (j,t) is:

$$(i,s) > (j,t) \text{iff} ((i > j \wedge \text{xor}(s[j],t[i])) \vee (i < j \wedge (s[j] = t[i]))$$

where $\wedge$ denotes logical and, $\vee$ denotes logical or, $\neg$ denotes logical negation, and xor denotes logical exclusive-or.

In the time-stamp system there are $n2^n$ possible labels. We can construct a new label (i, s) for a thread i, greater than the labels of all other threads, by constructing a string of bits s such that for any thread j with label (j, t)

if $i > j$ then $s[j] := \neg t[i]$
if $i < j$ then $s[j] := t[i]$
if $i = j$ then $s[i] :=$ anything.

Note that for the label (i, s), $s[i]$ is irrelevant, since a label never needs to be compared against itself.

Exercise 20. Give Java code to implement the Timestamp interface of Fig. **??** using unbounded labels. Then, show how to replace the pseudocode of the Bakery lock of Fig. **??** using your Timestamp Java code [**?**].

Solution See code folder for this chapter's solutions.

```
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 1: The Flaky lock used in Exercise 11.

```
1  public class TreeLock implements Lock
2  {
3    private class PetersonNode;
4    private final int _num_threads;
5    private final int _num_nodes_in_tree;
6    private final PetersonNode[] _tree_ary ;
7    final private ThreadLocal<Integer> THREAD_ID;
8
9    public TreeLock(int num_threads);
10   public void lock();
11   public void unlock();
12 }
```

Figure 2: Question 1.1 - class overview

```
1  public TreeLock(int num_threads) {
2    int fixed_num_threads=1;//find power-of-two sizes best matching "num_threads"
3    while (fixed_num_threads < num_threads)
4      fixed_num_threads <<= 1;
5    _num_threads = fixed_num_threads;
6
7    //build tree of locks, using array representaion.
8    final int tree_height = (int) (Math.log(_num_threads)/Math.log(2)) - 1;
9    _num_nodes_in_tree = (int) Math.pow(2, tree_height+1)-1;
10   _tree_ary = new PetersonNode[_num_nodes_in_tree];
11   int thread_indx_mask = 1 << tree_height;
12   int curr_level_end_indx = 1;
13   for(int i=0; i<_num_nodes_in_tree; ++i) {
14     if(i == curr_level_end_indx) {
15       thread_indx_mask >>>= 1;
16       curr_level_end_indx = curr_level_end_indx *2 + 1;
17     }
18     tree_ary[i] = new PetersonNode(thread_indx_mask);
19   }
20 }
```

Figure 3: Question 1.1 - constructor

```
1  public void lock() {
2    //deduce thread leaf index
3    int thread_id = THREAD_ID.get();
4    int curr_node_indx = _num_nodes_in_tree - (_num_threads/2) + (thread_id / 2);
5
6    //lock all nodes from the leaf to the root
7    while(curr_node_indx >= 0) {
8      _tree_ary[curr_node_indx].lock(thread_id);
9      if( (curr_node_indx - 1) < 0)
10       break;
11     curr_node_indx = (curr_node_indx - 1) / 2;
12   }
13 }
```

Figure 4: TreeLock class: lock() method

```
1  public void unlock() {
2    int thread_id = THREAD_ID.get();
3    int curr_node_indx = 0;
4
5    //go from the root to the lead and unlock
6    do {
7      if (thread_id != _tree_ary [curr_node_indx ]. locked_thread_id ())
8        throw new UnsupportedOperationException("Not supported yet.");
9
10     _tree_ary [curr_node_indx ]. unlock( thread_id );
11     final int next_son_left = 2*curr_node_indx + 1;
12     final int next_son_right = 2*curr_node_indx + 2;
13
14     if ( next_son_left < _num_nodes_in_tree) {
15       if (thread_id == _tree_ary[ next_son_left ]. locked_thread_id ())
16         curr_node_indx = next_son_left ;
17       else
18         curr_node_indx = next_son_right ;
19     }
20     else break;
21   } while(true);
22 }
```

Figure 5: Question 1.1 - unlock method

```
1  class FastPath implements Lock {
2    private static ThreadLocal<Integer> myIndex;
3    private Lock lock;
4    private int x, y = -1;
5    public void lock() {
6      int i = myIndex.get();
7      x = i;                          // I'm here
8      while (y != -1) {}              // is the lock free?
9      y = i;                          // me again?
10     if (x != i)                     // Am I still here?
11       lock.lock();                  // slow path
12   }
13   public void unlock() {
14     y = -1;
15     lock.unlock();
16   }
17 }
```

Figure 6: Fast path mutual exclusion algorithm used in Exercise 15.

```
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 ThreadLocal<Integer> myIndex;
7    private int last = -1;
8    int visit () {
9      int i = myIndex.get();
10     last = i;
11     if (goRight)
12       return RIGHT;
13     goRight = true;
14     if (last == i)
15       return STOP;
16     else
17       return DOWN;
18   }
19 }
```

Figure 7: The Bouncer class implementation.

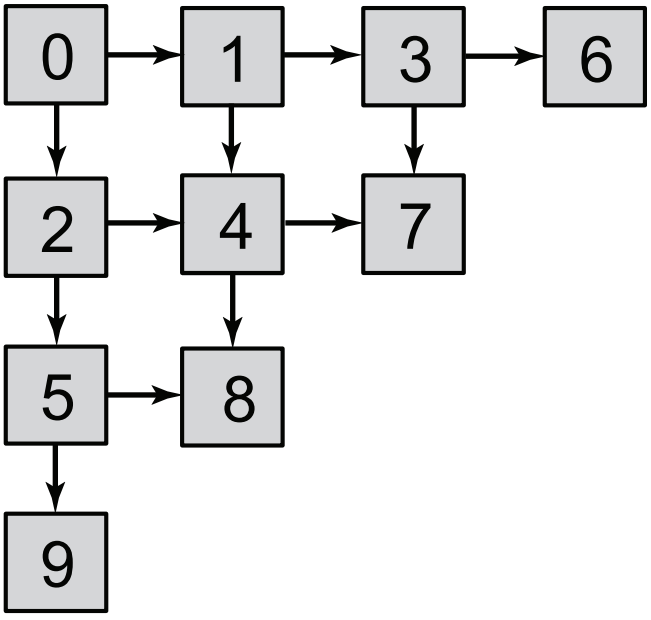

Figure 8: Array layout for Bouncer objects.