2.10 Exercises

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*?

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.

Exercise 11. Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 2.15 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?

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

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.

```
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 2.15 The Flaky lock used in Exercise 11.

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?

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.

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. 2.16. 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.

```
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 2.16 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; // initialize 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 2.17 The `Bouncer` class implementation.

Exercise 16. Suppose n threads call the `visit()` method of the `Bouncer` class shown in Fig. 2.17. 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.