Homework 02 — *COSC 273: Parallel and Distributed Computing*, Spring 2023

Due: Friday, 03/03/2023 at 11:59 pm

Instructions: You may work on this assignment in groups of up to 3 and submit a single solution for your group. All group members are responsible for understanding all submitted solutions.

Exercise 1. Consider the following modified `lock()` method for the Peterson lock (cf. Section 2.3.3 in *The Art of Multiprocessor Programming*):

```
public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    victim = i;
    flag[i] = true;
    while (flag[j] && victim == i) {} // wait
}
```

The only difference with the original method is that the statements `victim = i` and `flag[i] = true` are reversed in the modified version. Does the modified Peterson satisfy mutual exclusion? Why or why not? If so, you should argue that mutual exclusion is satisfied. If not, describe an execution for which mutual exclusion fails.

Exercise 2. So far in our discussion of locks, we have assumed that the only atomic operations to shared memory locations are `read` and `write`. Most modern computer hardware, however, supports other atomic operations as well. One such operation is the `getAndIncrement` operation which is implemented in Java for the `AtomicInteger` class. An `AtomicInteger` stores an integer, whose value can be accessed by the `get()` method. The `getAndIncrement` method (1) returns the current value of an `AtomicInteger`, and (2) increments the stored value as a single atomic operation. For example, if `AtomicInteger ai` stores a value `7`, and two threads (say, thread1 and thread2) concurrently call `ai.getAndIncrement()`, then exactly one thread will get the value `7`, and the other will get the value `8`. After both operations, the stored value will be `9`.

Using `AtomicInteger`s that support the `getAndIncrement` operation, devise a *simple* lock that works for any number of threads. Argue that your lock satifies mutual exclusion and starvation freedom.

Hint: consider Lamport's original idea of a ticket counter in a bakery.

Exercise 3. Consider the following `Bouncer` object:

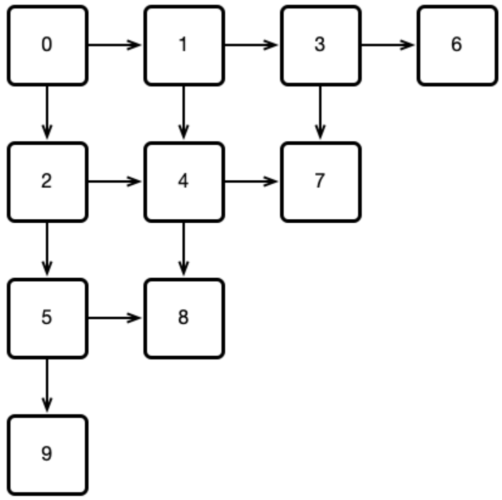
```
class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private boolean goRight = false;
    private int last = -1;
    int visit () {
        int i = ThreadID.get();
        last = i;
        if (goRight)
             return RIGHT;
        goRight = true;
        if (last == i)
            return STOP;
        else
            return DOWN;
    }
}
```

Suppose n threads call the `visit()` method. Argue that the following hold:

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

Exercise 4. So far in this course, we have assumed that all threads have IDs that are reasonably small numbers. In Java, however, thread IDs can be arbitrary `long` values. In this exercise, we will see how to use `Bouncer` objects as above to create unique IDs that are reasonably small compared to the number of threads.

Consider a 2D triangular array of ‘Bouncer‘ objects arranged as follows:



Suppose each thread performs the following procedure: All threads start by calling `visit()` on `Bouncer 0`. Whenever a thread visits a `Bouncer`, if the `Bouncer` returns `STOP`, the thread adopts the number of the `Bouncer` as its ID. If `DOWN` is returned, the thread then visits the `Bouncer` below; if `RIGHT` is returned, the thread visits the `Bouncer` to the right.

(a) If there are 2 threads, argue that all threads adopt unique IDs in the range 0 to 2.

(b) If there are 3 threads, argue that all threads adopt unique IDs in the range 0 to 5.

(c) More generally, argue that if there are n threads and sufficiently many `Bouncer` objects, then all threads adopt unique IDs. How many such `Bouncer`s are sufficient in this case?