EE 360P, TEST 1, Vijay K. Garg, Spring'17

NAME: Solutions

UT EID:

Honor Code: I have neither cheated nor helped anybody cheat in this test.
Signature:

Time Allowed: 75 minutes
Maximum Score: 75 points
Instructions: This is a CLOSED book exam. Attempt all questions.

Q.1 **(10 points)** Please give concise answers to the following questions.

(a, 2 point) What is busy waiting? How can busy waiting be avoided with the help of the thread scheduler of the system?

Busy waiting is a way of synchronization by repeatedly checking for certain condition to become true. It uses CPU time for without doing any useful work. It can be avoided by rescheduling the thread and moving it the queue of blocked threads. The thread is woken up by another thread when the condition may have become true.

(b, 3 pts) Add the synchronization code using binary semaphores to ensure that

- `bar2()` is not called untill `foo1()` has finished execution. and
- `foo3` and `bar2` are not executed concurrently i.e. either Thread 2 must wait for foo3() to finish before starting bar2, or Thread 1 must wait for `bar2()` to finish before starting `foo3()`.

Remember to show the initial value of your binary semaphores.

```
// declare your semaphores here
Semaphore s = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
```

```
Thread 1                              Thread 2
-------                               ---------

foo1();                                bar1();

s.release();                           s.acquire();
                                        mutex.acquire();

foo2();                                bar2();

mutex.acquire();                      mutex.release();

foo3();

mutex.release();                             bar3();
```

(c, 2 points) Consider the class shown below.

```
class Schedule {
    static int x = 5;
    static int y = 3;
    public synchronized int op1(){x = 7; return y-x;}
    public synchronized int op2(){y = 1; return 2*x + y;}
}
```

If one thread calls `op1` and another thread calls `op2` concurrently, then what values may be returned by `op1` and `op2`?

If op1() is executed before op2() we get -4 and 15.
If op2() is executed before op1() we get 11 and -6

(d, 3 points) What is meant by *resource ordering* in concurrent systems with multiple resources. Why is resource ordering useful?

Resource ordering is used in concurrent systems to avoid deadlocks when threads need multiple resources. If threads order all the resources and acquire locks only in increasing order, then the system cannot have deadlocks (due to hold-and-wait cycles for resources).

Q.2 **(5 points)** A computer program has a method that takes 20% of the time during a sequential execution and it cannot be parallelized. Suppose that your manager has asked you to buy a new parallel computer so that the same program can run three times as fast. What is the minimum number of cores you need in your computer to get that speedup. Justify your answer.

Let T_p be the time on parallel computer, T_s be the time with serial execution and n be the number of cores. Then, from Amdahl's law

$T_p \geq T_s * (1/5) + T_s * (4/5) * (1/n)$

Therefore,

$5n \geq n * T_s/T_p + 4 * T_s/T_p$

By using $T_s/T_p = 3$, we get,

$5n \geq 3n + 12$.

Hence, $n \geq 6$.

Q. 3 **(10 points)** Consider the class `AtomicInteger()` in `java.util.concurrent.atomic`. Suppose the class comes with the following methods:

```
AtomicInteger(int initialValue) // Creates a new AtomicInteger with the given initial value.
int     get() // Gets the current value.
void    set(int newValue) //Sets the current value to newValue.
boolean compareAndSet(int expected, int newValue)
//Atomically sets the value to newValue if the current value equals expected value.
```

Extend the class `AtomicInteger` to `MyInteger` that provides the following additional methods.

```
int     incrAndGet() //Atomically adds one to the current value and returns updated value
boolean testAndSet() //Atomically sets the value to 1 and
// returns true if the previous value was nonzero and false otherwise.
```

Your implementation must be *lockfree*.

```
import java.util.concurrent.atomic.*;
class MyAtomicInteger extends  AtomicInteger {
    public MyAtomicInteger(int val) { super(val);}
      public int incrAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
    public boolean testAndSet() {
        while (true) {
            int current = get();
            int next = 1;
            if (compareAndSet(current, next))
                return (current != 0);
        }
    }
   }
```

Q. 4 **(5 points)** The following code to acquire resources in the dining philosopher problem suffers from deadlock. In this code, all forks are binary semaphores initialized to 1.

Add the synchronization code by declaring one or more additional semaphore and using your semaphore(s) to remove the deadlock in the code by using the following strategy. A philosopher can request forks only if it is standing. He can sit down only after releasing forks. Use semaphores to ensure that at most four philosophers can stand at any point.

```
import java.util.concurrent.Semaphore;
class DiningPhilosopher implements Resource{
    int n = 0;
    Semaphore[] fork = null;
    Semaphore butler = null;
    public DiningPhilosopher(int initN) {
        n = initN;
        fork = new Semaphore[n];
        for (int i = 0; i < n; i++) {
            fork[i] = new Semaphore(1);
        }
        butler = new Semaphore(n-1);
    }
    public void acquire(int i) throws InterruptedException {
        butler.acquire();
        fork[i].acquire();
        fork[(i + 1) % n].acquire();
    }
    public void release(int i) {
        fork[i].release();
        fork[(i + 1) % n].release();
        butler.release();
    }
  }
```

Q.5 **(20 points)** A `CountDownLatch` provides the following methods:

- `void await()`: Causes the current thread to wait until the latch has counted down to zero
- `void countDown()`: Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

The initial value of the latch is provided via its constructor. Implement a class called `MyLatch` that contains a CountDownLatch and an integer `numWaiting` that maintains the number of threads waiting for the latch to reach zero. It provides an additional method `getWaiting()` that returns the number of threads that are waiting on the latch. Give an implementation of `MyLatch` with appropriate synchronization using Java monitors.

```
import java.util.concurrent.*;
public class MyLatch {
    CountDownLatch latch;
    int numWaiting;

    public MyLatch(int initialValue){
       latch = new CountDownLatch(initialValue);
       numWaiting = 0;
    }
    public void await() throws InterruptedException {
      //Causes the current thread to wait until the latch has counted down to zero

        synchronized(this) {
           numWaiting++;
        }
        latch.await();
        synchronized(this) {
           numWaiting--;
        }
    }

    //Decrements the count of the latch, releasing all waiting threads if the count reaches zero.
    public void countDown() {
        latch.countDown();
    }
    public synchronized int getWaiting(){
        return numWaiting;
    }
}
```

Q.6 **(25 points)** Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold c passengers, where $c < n$. The car can go around the tracks only when it is full. Passengers invoke board and unboard whenever they want to take a ride. The car invokes load, runCar and unload in a cycle. Write a public class `RollerCoaster` using Java Monitor to ensure the following constraints.

- Passengers cannot board the car until the car has invoked load.
- The car cannot depart until c passengers have boarded.
- Passengers cannot unboard until the car has invoked unload.
- Car invokes load only if all passengers have unboarded, i.e., unload method invoked by the car blocks until all passengers in the car have unboarded.

Do not worry about the efficiency of your solution. It only needs to be correct.

```
public class RollerCoaster {
   int num = 0; //number of passengers in the car
   final int LOADING = 0;
   final int RUNNING = 1;
   final int UNLOADING = 2;
   final int C = 5; //capacity of the car
   int state = LOADING;

    public synchronized void board() throws InterruptedException {
       while ((state != LOADING) || (num >= C))
                wait();
       num++;
       notifyAll();
    }

    public synchronized void unboard() throws InterruptedException {
       while (state != UNLOADING)
                wait();
       num--;
       notifyAll();
    }

    public synchronized void load() throws InterruptedException {
     state = LOADING;
     notifyAll();
    }

    public synchronized void runCar() throws InterruptedException {
        while (num < C) {
                wait();
        }
        state = RUNNING;
    }

    public synchronized void unload() throws InterruptedException{
        state = UNLOADING;
        notifyAll();
        while (num > 0) {
                wait();
        }
```

```
    }
}
```