EE 360P: Concurrent and Distributed Systems
Assignment 2

Instructor: Professor Vijay Garg (email: garg@ece.utexas.edu)

Deadline: 11:59 pm Feb. 14th, 2019

The submissions must be uploaded to the canvas by the deadline mentioned above. Please submit one zip file per team, with the name format [EID1_EID2].zip. You should use the templates downloaded from the course github (https://github.com/vijaygarg1/EE-360P.git).

1. (a) **(15 points)** A CyclicBarrier is a synchronization aid that allows a set of threads to wait for all other threads to reach a common point. Whenever a thread calls the method await, it gets blocked until the number of parties specified in the constructor have also called await. When the specified number of threads have called await, all threads are released.

   Your implementation must allow a CyclicBarrier to be reused after the waiting threads are released. So, a thread may call await a multiple number of times. You need to implement the following methods using *semaphores*:

```
public class CyclicBarrier {
  public CyclicBarrier(int parties) {
    // Creates a new CyclicBarrier that will release threads only when
    // the given number of threads are waiting upon it
  }
  int await() throws InterruptedException {
    // Waits until all parties have invoked await on this CyclicBarrier.
    // If the current thread is not the last to arrive then it is
    // disabled for thread scheduling purposes and lies dormant until
    // the last thread arrives.
    // Returns: the arrival index of the current thread, where index
    // (parties - 1) indicates the first to arrive and zero indicates
    // the last to arrive.
  }
}
```

   (b) **(15 points)** Implement `MonitorCyclicBarrier` using Java Monitor. Its interface is identical to `CyclicBarrier`. You are not allowed to use semaphores for its implementation.

2. **(35 points)** As you know, the Red River Showdown football game draws fans of UT and OU from everywhere to the Cotton Bowl. Naturally, some fans drink a lot of "apple juice," and patrons constantly stumble to the bathroom throughout the game. However, there is

only one bathroom shared between UT and OU fans. You have been tasked with designing a system conforming to the following rules:

(a) The bathroom size is limited and can hold no more than 5 fans at any time.

(b) The bathroom can only hold fans from one team at any time, otherwise a fight might break out. More specifically, if there is a single UT fan in the bathroom there can be no OU fans in the bathroom and vice versa.

(c) We require some sort of fairness in that a fan requesting the bathroom is blocked until all other fans preceding them have entered the bathroom, regardless of the team (just as if fans were waiting in a line for the bathroom).

Implement a Java class `FairUnifanBathroom` satisfying these rules using Java Monitors or ReentrantLock and Condition. It should support the following methods:

```
public class FairUnifanBathroom {
  public void enterBathroomUT() {...}
  public void enterBathroomOU() {...}
  public void leaveBathroomUT() {...}
  public void leaveBathroomOU() {...}
}
```

3. **(35 points)** Use Java ReentrantLock and Condition to implement a linked list based priority queue `PriorityQueue`. Each node in the queue has two fields: name (a String) and priority (an integer in the range 0..9). The order of the nodes is always kept sorted based on the priority with 9 as the highest priority and 0 as the least. Nodes with the same priority number are kept in the order of inserts. Furthermore, you should not use a global lock for the queue, but instead use the "hand-over-hand" locking technique discussed in class. Your `PriorityQueue` should support the following methods.

```
public class PriorityQueue {
  public PriorityQueue(int capacity) {
    // Creates a Priority queue with maximum allowed size as capacity
  }
  public int add(String name, int priority) {
    // Adds the name with its priority to this queue.
    // Returns the current position in the list where the name was inserted;
    // otherwise, returns -1 if the name is already present in the list.
    // This method blocks when the list is full.
  }
  public int search(String name) {
    // Returns the position of the name in the list;
    // otherwise, returns -1 if the name is not found.
  }
  public String getFirst() {
    // Retrieves and removes the name with the highest priority in the list,
    // or blocks the thread if the list is empty.
    }
}
```