CMSC 433 Section 0101
Fall 2016
Midterm Exam

Directions: Test is open book, open notes, closed electronics. Answer every question; write solutions in spaces provided. Use backs of pages for scratch work. Good luck!

Honor Pledge: “I pledge on my honor that I have not given or received any unauthorized assistance on this examination.”

(Signed) ____________________________________________

Please do not write below this line.

1. ________
2. ________
3. ________
4. ________
5. ________

SCORE ________
1. (20 points) MULTITHREADING, TESTING AND STATE DEPENDENCY

(a) (5 points) Give two reasons why a multi-threaded version of an application might run more slowly than a single-threaded version.

(b) (5 points) When testing a multi-threaded program, you observe different results from different runs of the same test. What might be an explanation for this behavior?

(c) (5 points) Explain why it is good practice to put your tests for concurrent programs inside loops.

(d) (5 points) The BlockingQueue interface in Java includes an operation, offer(), that uses balking to add an element to the end of the queue if possible, and another operation, put(), that uses guarded suspension to add an element to the end of the queue. Under which circumstances would you use one of these operations instead of the other, and why?
2. (20 points) LOCKING, VISIBILITY AND ATOMICITY

(a) (5 points) Explain why immutable objects in Java do not require any special care in order to be safely published.

(b) (5 points) Consider the following field declaration:

```java
public volatile long shared;
```

and assume further that thread $t_1$ is trying to assign value $v_1$ to `shared` while $t_2$ is trying to set value $v_2$. Is it guaranteed that `shared` will wind up with either $v_1$ or $v_2$ when the threads finish? Explain.

An alternative means to achieving thread-safety would be to require calling code to use client-side locking. The next two parts of this question refer to this approach.

(c) (5 points) Explain how method-call `obj.m()` would be implemented in this case.

(d) (5 points) Give one advantage and one disadvantage to using this approach. Explain your answer.
3. (20 points) OBJECT COMPOSITION

In class we studied a class of thread-safe bounded counters. The method headers for this class were the following.

```java
public class BoundedCounterThreadSafe {
    private int value;
    private int upperBound;

    public synchronized int current(); // Return value
    public synchronized void reset(); // Reset value to 0
    public synchronized boolean isMaxed(); // Return true if value is maxed out
    public synchronized void inc(); // Increment value if < upperBound
}
```

This question refers to the above class.

(a) (10 points) Suppose we wish to implement a class of roll-over counters, with the property that if `inc()` is called when the counter has its maximum value, the counter "rolls over" to 0, instead of remaining at its maximum value. Would the following be a reasonable use of delegation to ensure safety? Explain.

```java
public class RolloverCounter {
    private BoundedCounterThreadSafe c;
    ....
    public void inc() {
        if (c.isMaxed()) {
            c.reset();
        }
        else {
            c.inc();
        }
    }
    ....
}
```
(b) (10 points) Suppose we wish instead to implement a class of *blocking counters*; for such counters, an `inc()` operation should block only if the counter is maxed out, and proceed only when it is not. Is the approach below a good one? Explain why or why not.

```
public class BlockingCounter {

    private BoundedCounterThreadSafe c;
    ....
    public synchronized void inc() throws InterruptedException {
        while (c.isMaxed()) {
            wait();
        }
        c.inc();
        notifyAll();
    }
    ....
}
```
4. (20 points) CONCURRENT COLLECTIONS

(a) (5 points) Explain why an ArrayList that is passed as an argument to Collections.synchronizedList() should not be published.

(b) (5 points) Explain how lock-striping is used in the implementation of ConcurrentHashMap to improve concurrent access to hash maps.

(c) (5 points) Explain the difference between a fail-fast iterator and a weakly consistent iterator.

(d) (5 points) Under what circumstances would you consider using a CopyOnWriteArrayList object in an application you are developing? Explain.
5. (20 points) CODING

In this question you are asked to implement a thread-safe class of bounded counting semaphores. The interface for the class you should implement is the following.

```java
public interface Semaphore {
    public void acquire() throws InterruptedException;
    public void release() throws InterruptedException;
}
```

The meaning of these operations is as follows. A bounded counting semaphore maintains a field indicating how many “permits” it currently has available, as well as a maximum number of permits that can be granted. `acquire()`, which is a blocking operation, should wait until the requested number of permits is available, then reduce the number of permits accordingly. `release()` is also blocking; it waits until the semaphore can accept the number of permits that are being released, then increases the number of permits and returns.

On the bottom of this page and all of the next page, complete your implementation of the class by providing code for the constructor and for each of the operations mentioned above. You may use whatever strategy you wish. Note that you need to ensure that the invariant is maintained, and that the methods satisfy the specifications given for them in their comments.

```java
public class BoundedSemaphore implements Semaphore {
    private int permits;
    private int limit;

    // Invariant: 0 <= permits <= limit

    BoundedSemaphore (int limit) { // Initially, permits == limit
        permits = limit;
    }
}
```

[PROBLEM CONTINUES ON NEXT PAGE.]
/**
 * Blocks until n permits can be acquired, then reduces permit count.
 * Precondition: 0 <= n <= limit
 * Postcondition: Block until permits >= n, then reduce permits by n
 */
public synchronized void acquire (int n) throws InterruptedException {

}

/**
 * Blocks until n permits can be released, then increases permit count.
 * Precondition: 0 <= n <= limit
 * Postcondition: Block until permits + n <= limit, then inc. permits by n
 */
public synchronized void release (int n) throws InterruptedException {

} // end of BoundedSemaphore class