CMSC 433
Spring 2016
Final Exam

Directions: Exam is open book, open notes, open electronics (but electronics must be in airplane mode). Answer every question; write solutions in spaces provided. Use backs of pages if necessary, but clearly indicate when this is the case. By writing your name above, you pledge to abide by the University’s Honor Code:

“I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.”

Good luck!

Please do not write below this line.

1. _______  6. _______
2. _______  7. _______
3. _______  8. _______
4. _______  9. _______
5. _______  10. _______

SCORE_______
1. (10 points) True / False. Indicate whether each of the following statements is “True” or “False” by writing a “T” or “F” in the space provided.

(a) Blocking queues are examples of synchronizers.

(b) An `InterruptedException` can be thrown by a thread when it is in the `Blocked` state.

(c) When an `ExecutorService`'s `shutdown()` method is called, the unexecuted tasks in the work queue are returned.

(d) akka guarantees at-most-once message delivery.

(e) akka supervisor actors may either resume, restart or stop a failed child, or escalate the failure to their own supervisors.

(f) MapReduce processing is based on the `map` and `fold_left` list operations from functional programming languages such as OCaml.

(g) The Hadoop design features the sending of “data to computation” rather than “computation to data.”

(h) If a node in a cluster fails during the execution of a Hadoop application, the application must be restarted.

(i) Every execution of a Java program is sequentially consistent.

(j) The insertion routine for the non-blocking stack is an example of an obstruction-free non-blocking algorithm.
2. (10 points) Visibility and publication

(a) (3 points) Give an example of an operation in Java that is atomic but not necessarily visible.

(b) (3 points) Give an example of an operation in Java that is atomic and visible.

(c) (4 points) Give the requirements that a class must satisfy in order for its objects to be immutable.
3. (10 points) Thread-safety and object composition

(a) (5 points) Suppose a programmer wants to implement a class of thread-safe blocking queues that filter out “bad elements.” The idea is that these filtered queues prevent the insertion of these elements; the precise notion of badness is passed in as part of the constructor of the object. Specifically, assume there is an interface `Condition<E>` given as follows.

```java
public interface Condition<E> {
    boolean isGood (E element);
}
```

The idea is that `isGood(e)` indicates that element `e` is “good”, and hence may be inserted into the queue; dually, if `isGood(e)` is violated, then `e` is “bad” and should not be allowed into the queue. In the latter case, any attempt to insert such an element should be ignored.

Now consider the following implementation of class `FilteredArrayBlockingQueue`.

```java
public class FilteredArrayBlockingQueue<E> {
    private ArrayBlockingQueue<E> backingQueue;
    private Condition<E> condition;
    public FilteredArrayBlockingQueue (Condition<E> c, int bound) {
        ... 
    }
    
    public synchronized void put(E e) {
        if (condition.isGood(e)) { // Only insert if e is good.
            try {
                backingQueue.put(e);
            }
            catch (InterruptedException exn) { }
        }
    }
}
```

Is this implementation of `put()` satisfactory? Why or why not?
(b) (5 points) Delegation is an approach to thread safety in which a class is thread-safe even though its methods are not synchronized. What conditions must be satisfied in order for delegation to ensure thread safety?
4. (10 points) Synchronizers

Complete the implementation of the inc() and dec() methods of the following thread-safe class of bounded blocking counters. You may not use intrinsic locks, but you may use other synchronizers, including ReentrantLock objects and conditions. You may add additional private fields. For full credit, do not use signalAll().

```java
public class BoundedBlockingCounter {

    private int count; // Invariant: 0 <= count <= max
    private int max;
    // Add additional fields here as needed

    public BoundedBlockingCounter(int max) { ... }

    // inc() waits until count is < max then increments
    public void inc() throws InterruptedException {
    }
}
```
// dec() waits until count is > 0 then decrements
public void dec() throws InterruptedException {

}
5. (10 points) Tasks and executors

(a) (4 points) It has been remarked that an ExecutorService resembles a “virtual processor.” Give two characteristics of an ExecutorService that support this observation, and two that do not.

(b) (3 points) A developer wishes to use a fixed-size thread pool in a compute-intensive application and reasons as follows. (1) The machine on which the application is to run has 4 cores. (2) Since only 4 threads can be running simultaneously, restricting the number of worker threads in the thread pool to 4 is reasonable. The application subsequently deadlocks. Explain how this could happen.

(c) (3 points) Explain what the term work-stealing means in the context of Fork/Join thread pools.
6. (10 points) Performance

(a) (5 points) An application developer determined that on a machine with 8 cores, and with a desired utilization of 75%, the optimal number of threads to allocate for his/her application was 18. What is the anticipated ratio of wait to compute time for this application? Show your work.

(b) (5 points) One criticism of traditional, work-queue-based executors is that the work queue is a *hot spot*; each worker thread must access this same data structure whenever the worker thread is idle. Explain how Fork/Join queues help address this issue.
7. (10 points) Actors and akka

(a) (3 points) Message passing in akka is said to be “locally FIFO.” What does this mean?

(b) (3 points) Explain how a supervisor actor resumes a child actor.

(c) (4 points) Explain why message classes in akka Java must satisfy the `Serializable` interface.
8. (10 points) MapReduce and Hadoop

(a) (5 points) In each case below indicate whether the given task in Hadoop is the responsibility of a Master node or Worker node.
   i. Distributing jar-files containing MapReduce code.
   ii. Performing Map operations.
   iii. Performing Reduce operations.
   iv. Reassigning MapReduce operations in case of node failure.
   v. Sorting result files of Map operations.

(b) (5 points) Explain the differences between the local standalone, pseudo-distributed and fully distributed configurations of Hadoop.
9. (10 points) The Java Memory Model

(a) (3 points) Give the program-order execution for the `main()` given below.

```java
public static int a;
public static int b;

public static void main(String[] args) {
    a = 1;
    b = 2;
    if (a < b)
        a = b+1;
    else
        b = 4;
}
```
(b) (7 points) Consider the following program.

```java
public static int a;
public static int b;
public static Object l = new Object();

public static Runnable r = new Runnable() {
    public void run () {
        synchronized (l) {
            b++;
        }
        a = 1;
    }
};

public static void main(String[] args) {
    new Thread(r).start();
    synchronized(l) {
        b--;        
    }
    a = 2;
}
```

Prove that there is a race condition in this program by giving an execution showing that there are two conflicting events that are not related by happens-before. You must indicate which events these are, and explain why they are not related.
10. (10 points) Non-blocking algorithms

In class we studied the `CopyOnWriteArrayList<E>` class, which implements the `List<E>` interface (just like `ArrayList<E>` does). The class is thread-safe, but it does not use locking; the implementation strategy is that when the underlying `ArrayList<E>` is modified, a copy is first made, the modification made, and then the copy “committed” back.

Give an implementation of the `set()` operation for `CopyOnWriteArrayList<E>` that uses optimistic retrying. Your implementation may not use locking, but it may use `AtomicXXX` classes as needed. As part of your code, you must also provide an appropriate private field, called `backingList`, that holds the elements stored in the list.

The official description of the `set()` method is as follows.

<table>
<thead>
<tr>
<th>public E set(int index, E element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replaces the element at the specified position in this list with the specified element.</td>
</tr>
</tbody>
</table>

**Parameters:**
- `index` - index of the element to replace
- `element` - element to be stored at the specified position

**Returns:**
the element previously at the specified position

**Throws:**
- `IndexOutOfBoundsException` - if the index is out of range (index < 0 || index ≥ size())

Please provide your implementation on the next page. The following operations on `AtomicXXX` objects may be useful.

- `get()` Return the object stored in the `AtomicXXX` object.
- `compareAndSet()` Given in class.
class CopyOnWriteArrayList<E> {

    // Give declaration for backingList field here

    // Now give implementation of set() here

}