CMSC 433
Fall 2016
Final Exam

Directions: Exam is open book, open notes, closed electronics. Answer every question; write solutions in spaces provided. Use backs of pages if necessary, but clearly indicate when you do so. 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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

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) As long as an object is properly constructed (i.e. this does not escape during construction), the publication of the object is guaranteed to be safe.

(b) Iterators produced for ConcurrentHashMap objects are weakly consistent.

c) If f is a FutureTask object, then while method f.get() is being executed the thread state of f is BLOCKED.

d) If exec is an ExecutorService object, and method exec.shutdown() is called, then all waiting tasks in the work queue of exec are discarded (hence not run).

e) Thread-starvation deadlock cannot happen in executors that dynamically create new worker threads when all current worker threads are busy.

(f) akka guarantees locally FIFO message delivery.

(g) Every application-created ActorRef object in akka has a parent ActorRef object.

(h) The intermediate results produced by the Mapper part of a Hadoop application are stored in main memory when the map task is finished.

(i) The Java Memory Model guarantees that if a program is properly synchronized, then every execution of the program is sequentially consistent.

(j) You must use locking to implement operations that are based on optimistic retrying.
2. (10 points) Implementation concerns in multi-threading

(a) (4 points) Explain why thread-safety alone does not guarantee correct behavior of compound actions. What mechanism is typically used to ensure such actions are correct?

(b) (3 points) Compare and constrain balking and guarded suspension.

(c) (3 points) Why is notifyAll() generally preferred to notify() when one thread wishes to inform waiting threads that they should resume executing?
3. (10 points) Immutability and exceptions

(a) (5 points) Explain why immutable objects guarantee initialization safety but effectively immutable ones in general do not.

(b) (5 points) Explain why calls to Thread.sleep() must include mechanisms for handling with InterruptedException, while synchronized() { ... } blocks need no such mechanisms.
4. (10 points) Synchronizers

In class we studied the class `CountdownLatch` of latches. The key methods for objects in this class are:

`await()` Blocks until the internal counter maintained by the latch is 0, then terminates.

`countdown()` Reduces the value of the internal counter by 1, if the counter is greater than 0, and notifies threads blocked at `await()` if counter then reaches 0.

In this problem we will consider a generalization of `countdown()`, which takes an arbitrary integer as an argument (can be negative, 0, or positive) and subtracts that integer from the internal count, notifying threads blocked at `await()` if the value reaches exactly 0.

Complete the implementation for `GeneralCountdownLatch` given on the next page by providing code for the constructor, `await()` and `countdown()`. You may not use any monitor locks in your code; only `ReentrantLock` objects may be employed for synchronization purposes.
public class GeneralCountdownLatch {

    private int count;
    private ReentrantLock lock;
    private Condition cond;

    public GeneralCountdownLatch (int count) {

    }

    public void await() throws InterruptedException {

    }

    public void countdown(int delta) {

    }

}
5. (10 points) Tasks, executors and performance

(a) (4 points) Suppose you have an 8-core machine, and a stream of independent tasks to process (so no task will wait on the results produced by another task). You know that on average, each task spends 60% of its time waiting (i.e. to acquire locks it needs) and 40% actually computing. Furthermore, you are willing to devote 50% of your machines utilization to this application. How many threads should your fixed-size thread pool contain? Show your work.

(b) (3 points) The documentation for Fork/Join pools says that the associated executor has as many worker threads as there are available processors. Explain why this implies that that Fork/Join pools are intended for compute-intensive applications, in which tasks spend almost no time waiting.

(c) (3 points) It is often said that optimistic-retrying schemes, such as the one that underpins the CopyOnWriteArrayList class, work poorly when contention involving updates is high (in other words, when a number of different threads are trying to write at the same time). Explain why this is the case.
6. (10 points) Actors and akka

(a) (3 points) In traditional actor systems actors are single-threaded, and all interaction among actors is handled via message passing. Why do these factors imply that locking is not needed when programming with actors?

(b) (4 points) In the akka system actors (objects of type ActorRef) wrap a message handler (object of type UntypedActor) inside code that manages the mail box for the actor and also handles the fault-tolerance mechanisms in akka. Which of the four supervisor responses to a supervisee’s failure depends on this design choice in akka? Explain.

(c) (3 points) What is the name of the actor that is an ancestor of all actors in an actor system?
7. (10 points) MapReduce and Hadoop

(a) (3 points) In functional programming data structures are not modified after they are created, although new data structures can include pieces from previously created ones. In class we saw that the MapReduce paradigm is based on functional programming principles. Explain how the idea of “no modification of existing data structures”, a key component of functional programming, is reflected in the MapReduce framework.

(b) (4 points) Hadoop is able to tolerate faults related to node malfunctions in the clusters on which applications run. Explain how Hadoop master nodes detect, and manage the recovery from, the failure of worker nodes.

(c) (3 points) When is sorting conducted during the execution of a Hadoop application? What keys are used for the sorting?
8. (10 points) The Java Memory Model

(a) (5 points) Consider the following execution for a program with two threads: main and T. (S represents the synchronization sequence.)

<table>
<thead>
<tr>
<th></th>
<th>main</th>
<th>T</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨main, begin⟩</td>
<td>⟨T, begin⟩</td>
<td>⟨main, begin⟩</td>
<td></td>
</tr>
<tr>
<td>⟨main, write, a, 1⟩</td>
<td>⟨T, read, a, 0⟩</td>
<td>⟨main, start, T⟩</td>
<td>⟨T, begin⟩</td>
</tr>
<tr>
<td>⟨main, start, T⟩</td>
<td>⟨T, write, a, 1⟩</td>
<td>⟨T, end⟩</td>
<td></td>
</tr>
<tr>
<td>⟨main, end⟩</td>
<td>⟨T, end⟩</td>
<td>⟨T, end⟩</td>
<td>⟨main, end⟩</td>
</tr>
</tbody>
</table>

Is this execution allowed by the Java Memory Model? Explain.
(b) (5 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 () {
        b = 1;
    }
};

public static void main(String[] args) {
    new Thread(r).start();
    a = 1;
}
```

Is there a race condition in this program? Why or why not?
9. (10 points) Atomics and optimistic retrying

`compare-and-swap()` is an atomic operation supported by some hardware instruction sets. It takes three arguments: a memory location L, an expected value E, and a new value N. The semantics of `compare-and-swap(L, E, N)` is given by the following pseudo-code, where `tmp` is a temporary variable allocated only to the thread performing the `compare-and-set()` operation.

```plaintext
tmp = L;
if (L == E) L = N;
return tmp;
```

`compare-and-set()` is an operation related to `compare-and-swap()`; rather than returning the (old) value stored in L, it returns a boolean indicating whether or not the update took place.

(a) (5 points) Give pseudo-code for the `compare-and-set(L, E, N)` operation.

(b) (5 points) Explain how `compare-and-set()` can be implemented using `compare-and-swap()`. Specifically, give pseudo-code containing one call to `compare-and-swap()`, and referencing only the values E and N, that implements `compare-and-set()`.
In this problem you will complete the implementation of a Hadoop application that, given annual salary data in a given year for a number different people, computes the average annual salary for all the people whose data in contained in the original data set.

More specifically, the data set you will process contains as keys strings corresponding to a person’s name, and an integer reflecting the total the person received for given year (in dollars). What you should output is a single string, **Average Annual Salary**, and a single number showing the average annual salary over all people in the data set.

You must only provide the completed **Mapper** and **Reducer** classes on the next page; you do not need to worry about `main()` method, which you should assume is provided for you. You may find the following methods useful.

- `context.write(k, v)` Writes key v and value v into Context object context.
- `intw.get()` Returns the integer stored in IntWritable object intw.
public class AverageAnnualSalary {

    private class SalarySumMR extends Mapper<Text, IntWritable, Text, IntWritable> {
        public void map(Text name, IntWritable pay, Context context) throws InterruptedException, IOException {

        }
    }

    private class SalaryAverageMR extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce (Text noName, Iterable<IntWritable> values, Context context) {

        }
    }

    public static main (...) { ... }
}