CMSC 433
Fall 2015
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.

<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) Fail-fast iterators can throw `ConcurrentModification` exceptions.

(b) `SynchronousQueue` objects are 0-capacity queues.

(c) `ForkJoin` executors use work-stealing to improve performance.

(d) akka guarantees that every message sent by one actor to another will be received.

(e) Sequence diagrams and message sequence charts are often used to design the message-passing strategies underpinning actor-system implementations.

(f) Hadoop applications are designed to be run only on single machines.

(g) The Java Memory Model only allows one event sequence for a single-threaded application, once the inputs are fixed.

(h) The Java Memory Model guarantees that properly synchronized programs can only have sequentially consistent executions.

(i) Compare-and-swap and compare-and-set are different names for exactly the same operation.

(j) Non-blocking algorithms always perform better than those that use locking.
2. (10 points) Thread Safety.

(a) (5 points) In order to improve concurrent access while preserving thread-safety in a class s/he is designing, a student decides to make only methods that change object state synchronized; those that read the state use no locking. Is this a good idea? Explain.

(b) (5 points) The Java Monitor pattern requires fields to be private. Why is this needed in order to ensure thread safety?
3. (10 points) Concurrent Collections.

(a) (5 points) Under what circumstances would a programmer need to use client-side locking when accessing a concurrent thread-safe collection? Explain.

(b) (5 points) Explain why does the implementation of ConcurrentHashMap uses lock striping.
4. (10 points) Deadlock.

Using no more than 2–3 sentences for each, define the following.

(a) (2 points) Circular waiting.

(b) (2 points) Deadlock prevention via lock ordering.

(c) (2 points) notify()-caused deadlock.

(d) (2 points) Nested monitor lockout.

(e) (2 points) Thread starvation deadlock.
5. (10 points) Executors and Thread Pools.

(a) (5 points) Suppose an application uses a fixed-size thread pool, and that the tasks submitted to the thread pool / executor are independent (i.e. do not share any data structures). Can thread-pool starvation occur? Explain.

(b) (5 points) One design principle for executors holds that long-running tasks should be handled by one executor, while short-running tasks should be handled by a second, different executor. Why is this?
6. (10 points) Parallelization

Consider the following approach to computing the element-wise sum of two equal-length arrays $a_1$ and $a_2$; the result should be an array $r$ of the same length as $a_1$ and $a_2$ with the property that $r[i] = a_1[i] + a_2[i]$ for all $i$.

```java
public class ArraySum {
    public static volatile int nWorkers = 1000;
    private static int i;

    // Precondition: a1.length == a2.length
    public static int[] arraySum(int[] a1, int[] a2) {
        ExecutorService exec = Executors.newFixedThreadPool(nWorkers);
        int[] result = new int[a1.length];
        for (i = 0; i < result.length; i++) {
            exec.execute(new Runnable() {
                public void run () {
                    result[i] = a1[i] + a2[i];
                }
            });
        }
        exec.shutdown();
        try {
            exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
        } catch (InterruptedException e) { }
        return result;
    }
}
```

(a) (5 points) This code compiles and produces the correct result, but its performance is poor in general. Why is that?
(b) (5 points) How would you change this code to improve its performance? You do not need to rewrite the code; just explain how you would do so.
7. (10 points) Actors / akka.

(a) (5 points) Explain why messages in akka must be Serializable.

(b) (5 points) Explain the difference between the resume and restart approach to handling child failure in the akka supervision mechanism.
8. (10 points) MapReduce / Hadoop.

When a college professor is considered for tenure and promotion, one factor that committees making these decisions consider is the so-called *citation count*: how times are the professor’s scholarly works (books, papers, etc.) are cited by others?

In this problem you will implement a mapper and reducer that, when embedded appropriately in Hadoop, yields an application that, given a list of scholarly papers, computes the citation counts for authors whose works are cited in these papers.

The input to your application is a list of key/value pairs, where a key is `Text` object representing the title of a paper and a value is a `Text` object containing the full text of the paper, including the list of references. The output of your application is a list of key/value pairs, where each key is a `Text` object representing an author name and each value is an `IntWritable` object representing the number of times the author is cited in the list of references of the papers.

You may use the following method, which takes the text of a paper and returns a list of authors whose names appear in the reference list.

```java
public class CitationUtils {
    public static ArrayList<String> getCitedAuthors (String s) {...}
}
```

Complete the implementations of `MapClass` and `ReduceClass` given on the next page. You may find the following instance constructors / methods useful from `Text`:

- `public Text()` – creates an empty `Text` object
- `public Text(String s)` – creates a `Text` object containing `s`
- `public String toString()` – returns the string embedded in the `Text` object
- `public void set(String s)` – changes the contexts of the `Text` object to `s`

The `Context` instance method `write(key, val)` should also be used. You may add fields and other private methods if you wish.
public class MapClass extends Mapper<Text, Text, Text, IntWritable> {
    public void map(Text key, Text value, Context context) throws InterruptedException, IOException {
    }
}

public class ReduceClass extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce (Text key, Iterable<IntWritable> values, Context context) throws InterruptedException, IOException {
    }
}
9. (10 points) Java Memory Model.

(a) (5 points) Given the program-order event sequence for the execution of `main()` in the following code.

```java
public class Main {
    public static int a = 2;
    public static int b = 1;

    public static void main(String[] args) {
        a += b;
        b = a + 1;
    }
}
```

(b) (5 points) According to the JMM definition, what must hold in order for two events to be deemed as evidence that an execution of a program has a data race?
10. (10 points) Non-blocking Queues.

In class we saw examples of non-blocking stack and queue data structures. These data structures avoid using locks, while still ensuring thread-safety, using operations such as `compareAndSet()`.

In the case of queues, we studied an implementation of the operation `put()` in a class `LinkedQueue<E>`, which inserts a new element at the tail of the queue. In this problem, you are asked to implement the operation `take()`, which deletes and returns the element at the head of the queue, if the queue is non-empty. Otherwise, it returns `null`.

On the next page, fill in the implementation of this method. You may find the following instance methods from `AtomicReference<E>` useful.

```java
boolean compareAndSet (E expected, E new) 
    Compare the current contents of the AtomicReference object to expected, and if they are the same, update the contents to new, returning true. Otherwise, do nothing and return false.

E get() Return the contents of the AtomicReference object.
```

(Hint: can the `pop()` operation in `ConcurrentStack<E>` be adapted?)
public class LinkedQueue <E> {

    private static class QNode <E> {  // Class of nodes in linked list
        final E item;
        final AtomicReference<QNode<E>> next;

        public QNode(E item, QNode<E> next) {
            this.item = item;
            this.next = new AtomicReference<QNode<E>>(next);
        }
    }

    private final QNode<E> dummy = new QNode<E>(null, null);
    private final AtomicReference<QNode<E>> head
            = new AtomicReference<QNode<E>>(dummy);
    private final AtomicReference<QNode<E>> tail
            = new AtomicReference<QNode<E>>(dummy);

    public E take() {

    }

}