Name: ____________________________

CMSC 433 Section 0101  
Spring 2014  
Final Exam

Directions: Exam is closed book, closed notes. 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td>6.</td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>7.</td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>8.</td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>9.</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>10.</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” to the left of the statement.

(a) Every object in Java has an intrinsic / monitor lock associated with it.
F

(b) Reads and writes of 32-bit values are guaranteed to be atomic in Java.
T

(c) You should not let this escape during object construction.
F

(d) notify() rather than notifyAll() should be used when removing threads from wait-sets.
F

(e) If a class is thread-safe, then all compound actions a client might use are guaranteed to complete successfully.
T

(f) Explicit locks in Java are not re-entrant.
F

(g) If all tasks being processed by an Executor are independent, then thread-starvation deadlock is impossible.
T

(h) Recursive algorithms can be efficiently parallelized by having each recursive call generate a new task.
F

(i) In order to make a remote object available to clients, a server must register it with an object registry.
T

(j) If a worker node fails during the running of a job in Hadoop, the job is guaranteed to fail and must terminate.
F
2. (10 points) Java Memory Model

(a) (5 points) The Java Memory Model guarantees so-called *out-of-thin-air* safety. Explain what this means.

(b) (5 points) When does an execution of a Java program contain a data race, according to the Java Memory model?
3. (10 points) Visibility and Publishing.

(a) (5 points) Can constructors in Java be viewed conceptually as being atomic? Explain why or why not.

(b) (5 points) Explain the difference between immutable and effectively immutable objects. Why must so-called safe publication practices be used when publishing the latter but not the former?
4. (10 points) Object Composition.

   (a) (5 points) Explain what the Java Monitor Pattern is. Why does it ensure thread-safety?

   (b) (5 points) Suppose we wish to collect wait-time information for a so-called “linked blocking queue”, which is an unbounded queue that blocks threads trying to take elements when the queue is empty, just like any blocking queue. Apart from the possibility for overflow in the `waitTime` field, explain why the following approach to doing this is a very bad idea.

   ```java
   public class InstQueue<E> extends LinkedBlockingQueue<E>{
       private LinkedBlockingQueue<E> queue;
       private long waitTime;

       public synchronized E take() throws InterruptedException {
           long startTime = System.nanoTime(); // Set start time
           E result = queue.take(); // Set start time
           long endTime = System.nanoTime(); // Set end time
           waitTime += (endTime - startTime); // Update waiting time
           return(result);
       }
   }
   ```
5. (10 points) Synchronizers. In class we discussed *latches* in general, and Java’s *CountDownLatch* class in particular. Complete the implementation of the following lock-based version of this class.

```java
public class LockingCountDownLatch {
    private int count;
    LockingCountDownLatch (int count) {
        if (count < 0) throw new IllegalArgumentException();
        else this.count = count;
    }
    public synchronized void countDown() {
        // Complete this. If count is 0 do nothing. If count is > 0
        // decrement. If count then becomes 0, release waiting threads.
    }
    public synchronized void await() throws InterruptedException {
        // Complete this. If count is 0 do nothing and exit.
        // If count is > 0 then enter wait set.
    }
}
```
6. (10 points) Executors and Thread Pools.

(a) (5 points) Suppose you are designing a fixed-size thread pool for running independent tasks. The number of CPUs your computer has is 16, the desired utilization is 0.5 for the application, and the wait-to-compute ratio is 5/2. How many worker threads should your pool contain? Show your work.

(b) (5 points) Explain what work-stealing is in a Fork/Join thread pool. Why is it used?
7. (10 points) Concurrent Collections.

(a) (5 points) What is *lock striping*, and why does the implementation of `ConcurrentHashMap` use it?

(b) (5 points) In the implementation of `CopyOnWriteArrayList<E>`, instance methods that modify the state of the array are required to make a copy of the entire array, modify the copy, then use optimistic retrying to replace the old internal array with the new one. What operations become more efficient as a result? Under what circumstances would using a `CopyOnWriteArrayList<E>` make sense in an implementation?
8. (10 points) Remote Method Invocation.

(a) (5 points) Explain why objects passed as arguments to, or returned as results from, remote-method invocations must be **Serializable**.

(b) (5 points) Why must implementors of remote objects in the Java RMI framework ensure that the classes for these remote objects are thread-safe?
9. (10 points) Non-locking data structures.

(a) (5 points) Give a non-locking implementation of compareAndSet in terms of compareAndSwap. The header for compareAndSet should look like this:

    boolean compareAndSet (location l, value exp, value new).

(Hint: You should not need all the room below for your answer!)
(b) (5 points) Recall the following lock-free implementation of the \texttt{pop()} method for stacks, as discussed in class.

\begin{verbatim}
AtomicReference\langle Node\langle E\rangle \rangle top = new AtomicReference\langle Node\langle E\rangle \rangle();
...
public E pop() {
    Node\langle E\rangle oldHead;
    Node\langle E\rangle newHead;
    do {
        oldHead = top.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
}
\end{verbatim}

The class notes state that \texttt{compareAndSet()} provides visibility guarantees for the updates it performs that are analogous to those for \texttt{volatile} variables, and that without these guarantees the above algorithm would not be correct. Explain why this is the case; why does the correctness of the algorithm depend on these visibility guarantees?
Programming. Programs such as Google’s search engine make heavy use of URL indexing. A URL index lists, for a given URL, other URLs that have a link to that URL. This information can then be used to assess how popular a given site is, and hence how high up in a list of search results it should appear.

In this problem you will implement a mapper and reducer that, when embedded appropriately in Hadoop, yields an application that converts a list of URL / content pairs into an index showing, for a given URL, which URLs link to it.

More specifically, the input to your application is a list of key/value pairs, where a key is Text object containing a single URL and a value is a Text object containing the full HTML source at that URL. The output of your application is a list of key/value pairs, where each key is a Text object containing a URL and each value is a Text object containing a space-separated list of URLs having the property that the key appears in the HTML source associated to the URL.

For example, suppose an input key is www.umd.edu, and the associated value, the contents of the HTML value at www.umd.edu, has form “... www.bar.com ...” (in other words, the URL www.bar.com appears in the HTML). The output of the hadoop application should include www.bar.com as one of the keys, and the value associated to the key should be a string of form “... www.umd.edu ...” (this string is a list of space-separated URLs, in other words, with one of the URLs being www.umd.edu).

You may use the following method, which takes a string and returns a list of URLs found in the string.

```java
public class URLUtils {
    public static ArrayList<String> getURLs (String s) {...}
}
```

Complete the implementations of `MapClass` and `ReduceClass` below in such a way that a hadoop application can use them to generate an index. 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.
Finish the implementation of the class below and the one on the next page. You may add fields and other private methods if you wish.

```java
public class MapClass extends Mapper<Text, Text, Text, Text> {
    @Override
    public void map(Text key, Text value, Context context)
        throws InterruptedException, IOException {
    }
}
```
Finish the implementation of this class also.

```java
public class ReduceClass extends Reducer<Text, Text, Text, Text> {
    @Override
    public void reduce (Text key, Iterable<Text> values, Context context)
        throws InterruptedException, IOException { }
}
```