Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Spring 2010
May 18, 2010

Guidelines

• This exam has 12 pages (including this one); make sure you have them all. Several pages are designed for overflow and blank (except for a header stating that they are blank).

• You are not allowed to sit next to your friend, roommate, or the people you normally sit next to in class. Find a completely different neighborhood of the classroom to sit in.

• Put your name on the exam before starting the exam. Write your answers directly on the exam sheets.

• Bring your exam to the front when you are finished. Please be as quiet as possible.

• If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.

• One page of notes: You may bring one page of notes to use during the exam.

• You will be graded on only 10 out of 11 questions. Clearly mark which one question you want skipped.

• Each question is worth 10 points.

• You may avail yourself of the punt rule. In addition to skipping one question, you may punt as many as you wish. If you write down punt for a question, you will earn 3 points for that question. Make it clear what you want to punt on.

• Use good test-taking strategy: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

• Ignore InterruptedException: In all questions, you are free to ignore InterruptedException. You don’t need to worry about it, catch it, or deal with it.

• Abbreviate class names: You may abbreviate class names. For example, you can use IOE rather than IOException if it is reasonably obvious in context.

• If you need access to the Javadoc for any class in the JDK, raise your hand. I will either bring you an iPad with the Javadoc on it, or you may use a laptop at the front of class to browse the Javadoc.
Concurrency

1. **Atomicity Violation** Give an example of code that uses a ConcurrentHashMap and has a concurrency bug due to an atomicity violation, describe a specific example of an incorrect behavior that can result from the atomicity violation, and show how to fix the code using the methods provided by ConcurrentHashMap. Don’t worry about exact method names; any reasonable guess will be fine.
2. **AtomicInteger** Write a method that atomically doubles the value of an AtomicInteger, and returns the doubled value. Some of the methods for AtomicInteger are given below. The set given suffices to complete the question, although you do not need them all. You may also use any other method from the class:

```java
class AtomicInteger {
    int addAndGet(int delta) {...}
    int getAndAdd(int delta) {...}
    int get() {...}
    void set(int newValue) {...}
    int incrementAndGet() { ... }
    boolean compareAndSet(int expect, int update) {...}
    ...
}

public static int doubleAndGet(AtomicInteger a) {
```
3. **Fair locks** Several concurrent abstractions, such as locks, provide fair and unfair variants. For fair variants, requests are granted in a first-come, first-served order. With unfair variants, a request may be granted to one thread even if another thread is blocked waiting for a request to be granted. Assume the costs to perform the two variants are identical (e.g., the cost to acquire a fair or unfair lock are identical). In what ways might using unfair variants give better performance?
4. Multithreaded test case: Write a multithreaded test case for a CountDownLatch involving 3 threads. You want to show that if you initialize a CountDownLatch with a count of 2, and two threads await() on it, they both both blocked. If the third thread then calls countDown once, the threads are still blocked. If the third threads then calls countDown() a second time, both waiting threads are unblocked.

As a reminder of how the metronome timer and testing framework works, here is the public test from project 5 that checks that two locks provide mutual exclusion.

```java
static class TwoLocksProvideMutualExclusion extends MultithreadedTestCase {
    final Lock lock0, lock1;

    public TwoLocksProvideMutualExclusion(Lock lock0, Lock lock1) {
        this.lock0 = lock0;
        this.lock1 = lock1;
    }

    public void thread0() {
        lock0.lock();
        assertEquals(0, getTick());
        waitForTick(2);
        lock0.unlock();
        assertEquals(2, getTick());
    }

    public void thread1() {
        waitForTick(1);
        lock1.lock(); // should block here
        assertEquals(2, getTick());
        waitForTick(3);
        lock1.unlock();
        assertEquals(3, getTick());
    }
}
```
Security

5. Concisely describe the basic mistake that leads to web applications being vulnerable to Cross site scripting (XSS).

6. What is the first line of defense against Cross site request forgery? Perhaps not a perfect solution, but something easily to add to a web site that will at least make it significantly more difficult to execute a CSRF attack, if not block it all together.
Decorators, I/O

7. SlowOutputStream Decorator: You need to write a class that is a SlowOutputStreamDecorator. A SlowOutputStream is something you might, for example, want to use in a unit test or performance test where you wanted to simulate how something would perform if it was writing to a network connection that could only handle 100,000 bytes/second. A SlowOutputStream decorates an OutputStream, and when constructed two additional arguments that give the delay per byte written, specified by millisecondDelay and nanosecondDelay. You can sleep for the that much time using Thread.sleep(long millis, int nanos). Note: in practice, the sleep method isn't particularly accurate, particularly at sleeping for sub-millisecond intervals. But we'll ignore that for this question.

Your class should extend OutputStream, which is defined as:

```java
public abstract class OutputStream implements Closeable, Flushable {
    public abstract void write(int b) throws IOException;
    public void write(byte b[]) throws IOException {
        write(b, 0, b.length);
    }
    public void write(byte b[], int off, int len) throws IOException {
        for (int i = 0; i < len; i++)
            write(b[off + i]);
    }
    public void flush() throws IOException {
    }
    public void close() throws IOException {
    }
}

public class SlowOutputStream extends OutputStream {
    final OutputStream o;
    final long milliDelay;
    final int nanoDelay;

    public SlowOutputStream(OutputStream o, long milliDelay, int nanoDelay) {
        this.o = o;
        this.milliDelay = milliDelay;
        this.nanoDelay = nanoDelay;
    }
}
MapReduce and distributed computing

8. Why do all of the Mappers need to complete before we execute any calls to the

\[
\text{Reducer.reduce((InputKey key, Iterable<InputValue> values, Context context))}
\]

method?

9. In a real Hadoop or MapReduce implementation, the workers send back progress information to the Master (e.g., I’m now 65% done... I’m now 66% done...). Suggest how this information might be used.
10. Do Map tasks running on workers send all of their output keys and values to the Master node? Why or why not?

11. MapReduce is inspired by functional programming, and the Map and Reduce are supposed to be “functional” or “not have side effects”, such as changing static fields or writing output other than via Context. Why? What can go wrong if mappers and reducers have side effects?
Blank page, for overflow