433 Final Exam, Fall 2006

December 20th, 2006

Guidelines

Put your name on each page before starting the exam, and write your answer directly on the exam sheet.

Punting a Question

You may *punt* on any question. To *punt*, write *punt* for your answer, and you will receive 1/5 of the points for that question. If you already have written some material for the question when you decide to punt, be sure to make sure write down your decision to punt in such a way that it is very clear you have decided to punt the question. *The point of the punt rule is to avoid wasting both your time and my time when you realize that you don't think you can answer the question well.*

Blank extra pages

There are blank extra pages at the end of the exam. Feel free to use those if you need more space.

Grade

Question	Points	Grade
1	15	
2	15	
3	20	
4	15	
5	20	
6	15	

Questions

1. In the paper "A Note on Distributed Computing", the authors claim that some frameworks for distributed computing such as Remote Procedure Call (RPC) and CORBA offer an attractive vision that is not a good foundation on which to build reliable distributed systems.

   What is the vision that they warn against, and name and describe at least two of the differences (they mention 4) between local and distributed computing (you will lose points for listing differences not in their list of 4).

2. Write a class with two methods, t1() and t2(), that if called by two different threads, might deadlock. Add whatever fields or other methods you need. Your methods shouldn't (and don't need to) call wait() or await(). This question isn't asking to write some useful abstraction, but rather to provide a very concrete description of what deadlock is. Also provide a short (English) guideline for how to avoid deadlock (it doesn't matter if the guideline isn't universally applicable; just describe what steps suffice in the majority of cases to avoid deadlock).

3. This question concerns the difference between `Set<? extends Foo>`, `Set<Foo>` and `Set<? super Foo>`.

   (a) Describe some operations that will compile against some of them, but not against other, that distinguish them (e.g., you can do *Blah* to a X or a Y, but doing it to a Z won't compile. Give enough examples to distinguish all three.

   (b) Consider the two methods below:

```
class Bar {
  HashSet<Foo> foos;
  Set<? extends Foo> getFoosVersionOne() { return foos; }
  Set<? super Foo> getFoosVersionTwo() { return foos; }
  ... other methods
  }
```

   Assume your project is trying to decide which which return type the getFoos() method should have, based on what behaviors they want to encourage/discourage/permit/forbid in the caller of the getFoo() method. Summarize, briefly, how they should decide based on the behaviors they want to e/d/p/f.

   (c) Does the choice above actually forbid certain behaviors, or merely discourage them? Explain.

4. Short Answer:

   (a) What design pattern is used to construct instances of the InetAddress class, and why?

   (b) What design pattern is principally used in the Java input/output stream classes (e.g., java.io.DataInputStream)?

   (c) In distributed method invocation, such as our project 3, is a thread typically associated with a stub, a skeleton, or both?

5. Consider TwoPlaceQueue a blocking queue implementation that can store up to two elements. Write a multithreaded test case, using the metronome timer and testing framework discussed in class and used in project 2, that tests the scenario where an attempt to put a third item into the queue blocks because the queue is full. Your test case should only put a total of three items into the queue, remove all three items from the queue, and make sure the right items come out in the right order and that the queue is empty after all three items have been removed, and that the methods block when they should and don't block when they shouldn't.

   As a reminder of how the metronome timer and testing framework works, here is the public test from project 2 that checks that two locks provide mutual exclusion.

```
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());
    }
  }
```

   For your case, you should only worry about testing the methods shown below (take and put are the blocking methods). Note: unlike the BlockingQueue interface, in this class the put() and take() methods don't throw InterruptedException, and you don't have to worry about interrupts.

```
public class TwoPlaceBlockingQueue<E> implements BlockingQueue<E> {

  public E take() { ... }

  public void put(E e) { ... }

  public boolean isEmpty() { ... }

  ... other methods you shouldn't be testing
  }
```

Your MultithreadedTestCase for the scenario described:

6. Garbage collection: Assume you had to select just one garbage collection algorithm for a Java virtual machine: mark-and-sweep or a copying collector. What characteristic of your application would be most relevant to choosing the garbage collector? How would you choose (e.g., if my application required *Blarg*, I'd choose a *Snaff* garbage collector)?