

Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Fall 2004

December 13, 2004

Guidelines

Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand and I will come to you. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. Errors on the exam will be posted on the board as they are discovered. 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.

Question	Points	Score
1	12	
2	15	
3	18	
4	25	
5	30	
Total	100	

1. Short Answers (12 points, 2 each). Give very short (1 to 2 sentences) answers to the following questions. **Longer responses will not be read.**

(a) Ousterhaut argues that programmers should use event-based, rather than thread-based programming approaches, much more frequently than they do now. Describe the architecture of an event-based program.

(b) How is the value of the codebase system property used by a Java JVM?

(c) Synchronization in Java serves 3 functions. List and briefly explain each one.

(d) In class we talked about several ways to deal with state dependent actions. One way is called guarding; another is called retrying. Briefly compare and contrast these two approaches.

(e) Why, in Project 5, is the interface `FilterBuffer` a subtype of `RemoteFilterBuffer`? Why couldn't `RemoteFilterBuffer` be a subtype of `FilterBuffer` instead?.

(f) In project 5 we used a class called `ChickenPartType`. Its partial implementation is given below. What design pattern does this class implement?

```
public class ChickenPartType {  
  
    private ChickenPartType(String name) {this.name = name;}  
  
    public static ChickenPartType WING = new ChickenPartType("wing");  
    public static ChickenPartType DRUMSTICK = new ChickenPartType("drumstick");  
    public static ChickenPartType THIGH = new ChickenPartType("thigh");  
    public static ChickenPartType BREAST = new ChickenPartType("breast");  
}
```

2. Java Concurrency (15 points). Consider the following program.

```
public class ProducerConsumer {

    private boolean valueReady = false;
    private int bufferValue;

    void produce(int i) {
        while (valueReady) {}; // busy loop
        synchronized (this) {bufferValue = i; valueReady = true;}
    }

    int consume() {
        while (!valueReady) {}; // busy loop
        synchronized (this) {valueReady = false;return bufferValue;}
    }

    public static void main(String[] args) {
        final ProducerConsumer p = new ProducerConsumer();
//        3 THREADS CALL PRODUCE AND CONSUME ON P (see below)
//        .....
    }
}
```

Although this program is intended to follow normal buffer semantics, under some circumstances it will lose data.

Assuming 3 threads that loop forever, each one calling only p.produce() or only p.consume(). Provide a smallest program trace you can that shows how the above program can fail.

In describing your program trace, first briefly describe your 3 threads (i.e., are they producers or consumers). Then create a table like the one below, showing the steps executed by each thread. A step will be defined as an entry to p.produce's busy loop (written EPLP), an exit from p.produce's busy loop (written XPLP), an entry to p.consume's busy loop (written ECLP), an exit from p.consumer's busy loop (written XCLP), acquiring a lock on p (written AL), and releasing a lock on p (written RL). (The steps are interleaved, so only one thread should be taking a step on each line in the table.)

Thread 1	Thread 2	Thread 3	bufferValue
..	..		

Put your answer on the following sheet if necessary.

3. Java Concurrency (18 points) Some executions of the following program result in deadlock. Read the code carefully to determine how deadlock could occur. Then answer the questions that follow.

```
public class PointCount {
    static Point p1 = new Point(1, 1);
    static Point p2 = new Point(2, 2);

    public static class Point {
        int x, y;
        Point(int x, int y) {this.x = x;    this.y = y;}
        public int getX() {synchronized (this) {return x;}}
        public synchronized int getY() {synchronized (this) {return y;}}
        public void addPoints(Point p) {
            synchronized (this) {new Point(getX() + p.getX(), getY() + p.getY());}
        }
    }

    public class AddThread extends Thread {
        public void run() {
            while (true) {
                if (Math.random() < 0.5) {p1.addPoints(p2);}
                else {p2.addPoints(p1);}
            }
        }
    }

    public static void main(String[] args) {
        PointCount pc = new PointCount();
        PointCount.AddThread Th1 = pc.new AddThread();
        PointCount.AddThread Th2 = pc.new AddThread();
        Th1.start(); Th2.start();
    }
}
```

- (a) (5 points) Show a program trace that deadlocks. In describing your program trace, create a table like the one below, showing the steps executed by each thread. A step will be defined as an entry into a method (written Ename), where “name” is the name of the object and method (e.g., x.foo), an exit from a method (written Xname), acquiring a lock on an object (written Aobject), and releasing a lock on an object (written Robject). (The steps are interleaved, so only one thread should be taking a step on each line in the table.)

(b) (3 points) Draw a wait graph that illustrates the deadlock situation resulting from your execution. Remember that wait graphs show Threads as circles, objects as rectangles, lines from Thread t to Object o when t holds a lock on o , and lines from Object o to Thread t when t is blocked while trying to acquire the lock on o .

(c) (10 points) The `addPoints()` method must return the addition of two valid Points. Write a new version of `addPoints()` that maintains this behavior, but doesn't deadlock. You only need to add to or change a few lines in `addPoint()`. You may not change any other method or field of `PointCount`.

4. Java Concurrency (25 points). Fill in the code below. This application simulates customers going into a BarberShop to get a haircut. The barber works in a separate room. Customers wait in the waiting room separated from the barber's room by a door.

Customers enter the waiting room by calling `needACut()`. Each customer will have their own preferences on how they want their hair cut. These instructions are represented as a `Runnable` and are passed as a parameter to `needACut()`.

There are 4 seats in waiting room. Customers who want a haircut will take a seat and wait they're turn for the barber. Taking a seat is represented as storing the customer's `Runnable` instructions in a `LinkedList` called `waiters`. Assume that Customers can arrive at any time.

If a customer enters and all seats are full, they must leave. In this situation `needACut()` will return an `Exception` called `TooBad`.

The barber cuts clients' hair on a first-come, first-served basis. Each haircut will take an unpredictable amount of time. Haircuts are performed by calling `cut()`. `Cut()` takes the longest waiting customer from the waiting room. Then it invokes the `Runnable` representing the customers desired haircut. Note – make sure that customers can enter the waiting room, while the barber is cutting hair. That is, hair cutting shouldn't interfere with the customer's entering/leaving the waiting room.

If there are no customers waiting for a haircut when the barber looks in the waiting room, he takes a siesta. He is a heavy sleeper (some say he partakes of the demon alcohol – but you didn't hear that from me, hmm?) Because of this, a client who enters the BarberShop and finds no other clients waiting, will have to wake up the barber.

Fill in the following classes consistent with the description above. You may not add any variables of change the method interfaces. You may want to use the following `LinkedList` methods.

LinkedList methods:

```
public void addLast(Object o);  
public Object removeFirst();  
public int size();
```

```
import java.util.*;  
public class BarberShop {  
    private int maxBufferSize = 0;  
    private LinkedList waiters = new LinkedList();  
  
    public BarberShop (int size) {maxBufferSize = size;}  
  
    public static void main(String[] args) { // DON'T MODIFY  
        final BarberShop b = new BarberShop(4);  
  
        (new Thread () { // BARBER'S THREAD  
            public void run () {  
                while (true) {b.cut();}  
            }  
        }).start();  
  
        // CUSTOMERS THREAD - GENERATES CUSTOMERS ARBITRARILY  
        // EACH CUSTOMER CALLS  
  
        b.needACut(SomeRunnableObject);  
    }  
}
```

(over)

```
// FILL IN THESE 2 METHODS
```

```
    public void cut() {
```

```
    }
```

```
    public void needACut (Runnable job) throws TooBad {
```

```
    }
```

```
} // end BarberShop
```


5. RMI (30 points). The following classes do a simple iterative approximation of function integration. Since these approximations can be compute intensive, you've decided to use RMI so that the computations can be run on fast machines.

Consider the following classes.

```
public interface Evaluatable {
    public double evaluate(double value);
}

public interface RemoteIntegral extends Remote {
    public double sum(double start, double stop, double stepSize,
        Evaluatable evalObj) throws RemoteException;

    public double integrate(double start, double stop, int numSteps,
        Evaluatable evalObj) throws RemoteException;
}

public class RemoteIntegralImpl extends UnicastRemoteObject
    implements RemoteIntegral {

    public RemoteIntegralImpl() throws RemoteException {}

    // OTHER METHODS ELIDED TO SAVE SPACE
}

// An example function to be integrated
class Sin implements Evaluatable, Serializable {
    public double evaluate(double val) {
        return(Math.sin(val));
    }
}
```

The drivers for this application are called `RemoteIntegralClient` and `RemoteIntegralServer`. `RemoteIntegralClient` acquires a `RemoteIntegral` instance and uses it to integrate a function remotely (in this case $\sin(x)$). `RemoteIntegralServer` acts as a server for the `RemoteIntegral` class. The name of the host on which the server is running is passed to `RemoteIntegralClient` via the command line.

All RMI setup work must be done internal to the program (not on the command line). Assume that necessary class files are to be downloaded on the default http port from <http://www.classFilesToGo.com/WebServer> (for the server's classes) and www.classFilesToGo.com/WebClient (for the client's classes). `RemoteIntegralClient`'s files are in a directory called `DevClient` and `RemoteIntegralServer`'s files are in a directory called `DevServer`. The development directories are not accessible to each other. Assume that there's a `java.policy` file in the both the client and server's local directories.

Assume that the implementation of `RemoteIntegral` is not in the classpath of `RemoteIntegralClient` and that the implementation of any function to be integrated is not in the classpath of `RemoteIntegralServer`.

(a) (20 points) Fill in the code for RemoteIntegralServer and RemoteIntegralClient.

```
public class RemoteIntegralServer {  
    public static void main(String[] args) {
```

```
    }  
}
```

```
public class RemoteIntegralClient {  
    public static void main(String[] args) {
```

```
        remoteIntegral.integrate(0.0, Math.PI, 10000, new Sin());
```

```
    }  
}
```


- (b) (10 points) List exactly the class files that **MUST** be in the following four directories: WebClient and WebServer on www.classFilesToGo.com, DevClient and DevServer. You will be penalized heavily for adding unnecessary files.