Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Spring 2009
May 18, 2009

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. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

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

NOTE: There are quite a few questions to answer. Get started right away. Read each question carefully and answer the question that has been asked. Budget your time wisely. If you can’t answer a question in a few minutes, move on and come back to the difficult question later. Do not get bogged down on a single question.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Short answers (9 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. **Longer responses to these questions will not be read.**

(a) Synchronization in Java serves three functions: visibility, ordering, atomicity. Briefly explain each function.

**Answer:**

*Answer: i. Atomicity. Allows operation to be done without interference from other threads. ii. Visibility. Signals when values will be updated across threads iii. Ordering. Allows programmers to ensure that an operation A occurs before operation B*

(b) In the context of Java monitors, briefly define “barging”. **Answer:**

*Barging occurs when a thread waiting for a lock outside the monitor gets the lock before a signaled thread inside the monitor reacquires the lock.*

(c) In class we compared threads to events. Provide one strength and one weakness of threads. Provide one strength and one weakness of events.

**Answer:**

*Threads: Pro - allow use of multiple CPUs Con - locking can negatively affect performance
Events: Pro - run fast on single CPU because no context switches needed Con - can’t have long running handlers*
2. Java Concurrency (16 points). Consider the following program which is meant to implement a thread-safe 1-slot buffer.

```java
public class OneSlotBuffer {
    private boolean isFull = false;
    private int val;

    void put(int i) {
        while (isFull) {}; // busy loop
        synchronized (this) {
            val = i;
            isFull = true;
        }
    }

    int take() {
        while (!isFull) {}; // busy loop
        synchronized (this) {
            isFull = false;
            return val;
        }
    }

    public static void main(String[] args) {
        final OneSlotBuffer p = new OneSlotBuffer();
        // 3 THREADS CALL PUT AND TAKE ON P (see below)
        
    }  
}
```

(1) Although this program is intended to follow normal buffer semantics, under some circumstances a single data item can be read multiple times from the buffer.

Assuming 3 threads that loop forever, each one calling only `p.put()` or only `p.take()`. 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., do they put or take). Then create a table showing the steps executed by each thread. Be explicit in showing each step. Your presentation should interleave the steps so that only one thread takes a step on each line in the table.

(2) This program might also deadlock. Give a short description how this might happen.
Answer:

1) T1 does put()
T2, T3 do take()
T1 does exit put - isFull is true
T2 and T3 exit busy loop
T2 acquires lock and completes
T3 acquires lock and completes

2) This question was not counted in your grades.
Access to isFull is unsynchronized. T1 could set isFull to true, but this change might not be seen by T2 or T3.
3. Deadlock (10 points). The following program acquires locks on a set of resources. Once all locks are acquired a Runnable command is executed. The main method shows how the class is intended to work. Unfortunately, this program is not thread safe and may deadlock. Produce an alternative main() method and give an execution trace showing how it can lead to deadlock.

```java
public class ResourceMgr {
    Runnable comm;
    
    ResourceMgr (Runnable comm) {
        this.comm = comm;
    }

    public class Resource {
        public synchronized void acquire(List<Resource> rest) {
            if (rest.size() > 0) {
                (rest.remove(0)).acquire(rest);
            } else
                comm.run();
        }
    }

    public static void main(String[] args) {
        ResourceMgr bb = new ResourceMgr(new Runnable () {public void run () {}});
        // bb.new Resource() makes a new Resource inside bb
        final ResourceMgr.Resource r1 = bb.new Resource();
        final ResourceMgr.Resource r2 = bb.new Resource();
        final ResourceMgr.Resource r3 = bb.new Resource();
        List <Resource> resList = new ArrayList<Resource> ();
        resList.add(r2);
        resList.add(r3);
        r1.acquire(resList);
    }
}
```
Answer:

```java
public static void main(String[] args) {
    ResourceMgr bb = new ResourceMgr(new Runnable() {
        public void run() {}
    });
    final ResourceMgr.Resource r1 = bb.new Resource();
    final ResourceMgr.Resource r2 = bb.new Resource();

    final List<Resource> resList1 = new ArrayList<Resource>();
    resList1.add(r2);

    final List<Resource> resList2 = new ArrayList<Resource>();
    resList2.add(r1);

    Executor executor = Executors.newCachedThreadPool();
    executor.execute(new Runnable() {
        public void run() {
            r1.acquire(resList1);
        }
    });
    executor.execute(new Runnable() {
        public void run() {
            r3.acquire(resList2);
        }
    });
```
4. Optimistic Retry (15 points). Suppose your application needs to keep an up to date count of the number of threads using a resource and needs to do lengthy computations whenever a thread starts or stops using the resource (you will ignore these lengthy computations in your answer, however). Suppose further that you want to minimize overall locking time. Fill in the following code snippets that implement the Optimistic Retry strategy when threads start and stop using the resource. You may not add any synchronized blocks beyond that provided by numInside() and commit() methods.

```java
class Resource {
    protected Integer numCust = 0;
    public synchronized Integer numInside() {return numCust;}
    // FILL IN THE NEXT 3 METHODS ON FOLLOWING PAGE
    private synchronized boolean commit(Integer assumed, Integer desired);
    public void enter();
    public void exit()
    public static void main(String[] args) throws InterruptedException {
        final Resource r1 = new Resource();
        ExecutorService executor = Executors.newCachedThreadPool();

        Runnable r = new Runnable() {
            public void run() {
                r1.enter();
                /* Use Resource */
                r1.exit();
            }
        };
        executor.execute(r);
        executor.execute(r);
    }
}

private synchronized boolean commit(Integer assumed, Integer desired) {
    // FILL IN

    Answer:

    private synchronized boolean commit(Integer assumed, Integer desired) {
        if (numCust == assumed) {
            numCust = desired;
            return true;
        } else return false;
    }
}
```
public void enter() {
    // FILL IN

    Answer:

    public void enter() {
        boolean success = false;
        do {
            Integer old = numInside();
            Integer desired = new Integer(old + 1);
            success = commit(old, desired);
        } while (!success);
    }

    public void exit() {
    // FILL IN

    Answer:

    public void exit() {
        boolean success = false;
        do {
            Integer old = numInside();
            Integer desired = new Integer(old - 1);
            success = commit(old, desired);
        } while (!success);
    }
5. RMI. (20 Points). Write an RMI application that allows a central computer to upload statistics
gathering software to multiple remote host computers, which can execute the uploaded statistics
code and return the results to the central computer.

```java
public interface Statistics <T> {
    T compute();
}

// trivial implementation of Statistics
public class FileCounter implements Statistics<Integer>, Serializable {
    public Integer compute() {
        return number_of_files_on_host;
    }
}

Hosts that can accept statistics gathering code do so by running HasStatsImpl, which implement
the HasStats interface. The HasStatsImpl code registers a reference to itself in a local registry
under the name “StatsServer”.

```java
public interface HasStats extends Remote {
    public <T> T getStats(Statistics<T> st) throws RemoteException;
}

(a) Fill in the code needed to implement HasStatsImpl. It should start the registry on the default
port from within HasStatsImpl and should do all necessary setup activities needed so that
remote computers can later access it.
public class HasStatsImpl implements HasStats {
    public <T> T getStats(Statistics<T> st) {
        return st.compute();
    }
}

public static void main(String[] args) {

    Registry reg;
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        reg = LocateRegistry.createRegistry(1099);
        HasStats statsServer = new HasStatsImpl();
        HasStats stub = (HasStats) UnicastRemoteObject.exportObject(statsServer, 0);
        String objectURL = "StatsServer";
        reg.rebind(objectURL, stub);
    } catch (RemoteException e) {}
(b) Fill in the code needed to implement the RMI Client implementation called Crawler. Assume a HasStatsImpl is running on each remote machine under the name “StatsServer”. Assume that the list of remote machines on which statistics are to be gathered is passed in on the command line.

```java
public class Crawler {
    public static void main(String[] args) {
        try {
            for (String host : args) {
                Registry reg = LocateRegistry.getRegistry(host);
                HasStats server = (HasStats) reg.lookup("StatsServer");
                Statistics<Integer> st = new FileCounter();
                Integer stat = server.getStats(st);
                System.out.println(stat);
            }
        } catch (RemoteException e) {} 
        } catch (NotFoundException e) {} 
    }
}
```

Answer:

```java
public static void main(String[] args) {
    try {
        for (String host : args) {
            Registry reg = LocateRegistry.getRegistry(host);
            HasStats server = (HasStats) reg.lookup("StatsServer");
            Statistics<Integer> st = new FileCounter();
            Integer stat = server.getStats(st);
            System.out.println(stat);
        }
    } catch (RemoteException e) {} 
    } catch (NotFoundException e) {} 
}
```

(c) What are the minimum files needed at the client? What are the minimum files needed at the server? What are the minimum files that must be made web accessible? For each such file is it in the clients codebase or the servers codebase or both?

Answer:

On Client: Crawler.java, FileCounter.java, HasStats.java, and Statistics.java
On Server: HasStats.java, HasStatsImpl.java and Statistics.java
Web-accessible: FileCounter.java (in client codebase)
MapReduce (10 points). Sketch out a Map Reduce function that reads a set of email messages and produce one list of all participants in each email thread. To do this you will need to pseudocode the Map function and the Reduce function. Remember that Map and Reduce are defined as follows:

Map (in_key, in_value) → list(out_key, intmed_value)
Reduce (out_key, list(intmed_value)) → list(out_value)

Assume that the inputs to Map are 1) the name of a mailbox file and 2) the contents of the mailbox file, in that order. You can also assume the availability of a library for extracting mail records from the mailbox file. These library functions include an iterator (not shown) that produces individual mail records (one for each email message) of type MailRec.

class MailRec {
    String getFrom (); // mail sender
    List<String> getTo (); // list of mail recipients
    ID getSubjectID (); // common ID for related subject lines
}

Map (in_key, in_value) {
    // FILL IN
    for (MailRec msg : in_value) {
        ID id = msg.getSubjectId();
        emitIntermediate (id, msg.getFrom());
        for (String to : msg.getTo())
            emitIntermediate (id, to);
    }
}

Reduce (out_key, list) {
    // FILL IN
    Answer:
    Set<String> all = new HashSet<String> ();
    for (String par : list) {
        all.add(par);
        emit (all);
    }
}
7. Java Synchronizers (20 points). The following class uses Java Semaphores to synchronously send messages between two threads. That is, the process of sending and receiving a message must be completed before any following messages are sent. Fill in the following code snippets to implement this behavior.

class Handshake {
    private Object local;
    Semaphore semin = new Semaphore(0);
    Semaphore semout = new Semaphore(0);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        final Handshake sync = new Handshake();
        Runnable receiver = new Runnable() {
            public void run() {
                while (true) {
                    System.out.println('Receiver has received ' + sync.receive());
                }
            }
        };
        Runnable sender = new Runnable() {
            public void run() {
                while {
                    sync.send(new Object);
                    System.out.println('Sender has sent message');
                }
            }
        };
        executor.execute(sender); executor.execute(receiver);
    }
}
public void send(Object x) {
    // FILL IN

    local = x;
    semin.release();
    try {
        semout.acquire();
    } catch (InterruptedException e) {} 
}

public Object receive() {
    // FILL IN

    try {
        semin.acquire();
    } catch (InterruptedException e) {}
    Object tmp = local;
    semout.release();
    return (tmp);
}