

Exam 2

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

December 3, 2002

Guidelines

Put your name and class account number on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. I will not accept exams until I ask for them. If you finish early use the time to recheck your answers. Please be as quiet as possible.

I will not take any questions during the exam. 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	25	
2	24	
3	25	
4	26	
Total	100	

1. (25 points) Short Answer (a sentence or two should be sufficient).

(a) (4 points) How do locks (when used correctly) prevent data races?

by ensuring that only one thread can execute code that would modify shared state.

(b) (4 points) In addition to preventing data races, what important feature do Java locks provide?

They ensure that changes to shared data in one thread are made visible to other threads

(c) (6 points) During their lifetime, threads can be in a number of possible states; name two of them.

Two of created, runnable, blocked, sleeping, or terminated.

(d) (6 points) A *state-dependent action* within a method requires the object to be in an acceptable state before the action can be performed. For example, to return a value, a queue's `dequeue` method requires the queue to be nonempty. Explain two ways that a state-dependent method can respond when the object is *not* in an acceptable state.

Two of:

- balking (e.g. fail on the attempt by throwing an exception)
- failing silently (e.g. just return without doing anything)
- wait until the condition is true
- wait for a fixed period, then balk
- balk and retry, or wait for a fixed time and retry,

(e) (5 points) Explain how locks create the tension between *safety* and *liveness* in multi-threaded programs.

Using locks ensures safety by preventing data races. However, locks can also cause deadlock, which inhibits the program from making progress, and thus threatens liveness.

2. (Code Analysis - 24 points) The following code snippets can display undesirable behavior in one or more ways and/or in one or more situations. Discuss very briefly what is wrong with each one and describe an example execution trace(s) that demonstrates the error's presence. Note: Try/catch blocks have been removed to save space. So assume that they are there.

- Example 1:

```
// count is always between 0 and MAX inclusive
1: class Ex1
2: {
3:     private int count = 0;
4:     private final int MAX = 10;

5:     public synchronized void inc ( ) {
6:         if (count >= MAX) {
7:             wait();
8:         }
9:         count++;
10:        notifyAll();
11:    }

12:    public synchronized Object dec( )
13:    {
14:        if (count <= 0) {
15:            wait();
16:        }
16:        count--;
17:        notifyAll();
18:    }
19: }
```

Count can go outside bounds. Ex. count is 0. T1 calls anEx1.dec(). T1 waits. T2 calls anEx1.dec(). T2 waits. T3 calls anEx1.inc(). T3 calls notifyAll(). T2 wakes up and decrements count. T1 wakes up and decrements count. count is now -1.

- Example 2:

```
// doCall() is defined elsewhere

1: class Ex2 {
2:     boolean stopFlag = false;
3:     private Thread t =
4:     new Thread() {
5:         public void run() {
6:             while(!stopFlag) {
7:                 doCall();
8:             }
9:         }
10:    };

11:    public void stop()
12:    {
13:        stopFlag = true;
14:    }
15: }
```

This question was thrown out due to ambiguity. The intention was that stopFlag was not visible to the thread because stop() and run() do not synchronize on Ex2.

- Example 3:

```
1: class Ex3 {
2:     public Log lookupLog(Map map, String name) {
3:         Log log = (Log) map.get (name);
4:         if (log == null) {
5:             log = new LocalLog(100);
6:             map.add(name,log);
7:         }
8:     return log;
9: }
10: }
```

Even when map is synchronized, you can have conflicting writes and read/write. Ex. T1 calls anEx3.lookupLog. name is not in map. map.get returns null. new Log created. T1 gets swapped out. T2 calls anEx3.lookupLog with same name as T1 had. T2 completes after inserting a new log into map. T1 resumes and inserts its map, overwriting the one inserted by T2.

3. (Threaded Programming - 25 points). Implement a class `MutexLock` having the following signature:

```
public class MutexLock {
    public synchronized void acquire();
    public synchronized void release() throws BadReleaseException;
    public synchronized boolean attempt();
}
```

The idea is to implement your own kind of lock that is slightly different from Java's built-in locks:

- To acquire a lock, the program would call `acquire()`. This will block until the thread can acquire the lock (i.e. until the thread currently holding it releases it). *A thread can only hold a lock once.* That is, if a thread calls `acquire()`, and then calls `acquire()` again, it will block.
- To release a lock, the program would call `release()`; this method will throw `BadReleaseException` if a thread other than the one holding the lock tries to release it.
- To *attempt* to acquire a lock, the program would call `attempt()`. If the lock is held by another thread, then `attempt()` will immediately return `false`, otherwise it will acquire the lock and return `true`.

Recall that the method `Thread.currentThread()` returns the `Thread` identifier of the currently running thread.

Answer:

```
public class MutexLock {
    private Thread holder = null;

    /* Just like Java---block until acquiring the lock; unlike Java,
       the same thread cannot hold the lock more than once. */
    public synchronized void acquire() {
        while (holder != null) {
            try { wait(); } catch (InterruptedException e) { }
        }
        holder = Thread.currentThread();
    }

    /* Just like Java---release held lock; throws an exception if the
       wrong thread tries to release the lock */
    public synchronized void release() throws BadReleaseException {
        if (holder != Thread.currentThread())
            throw new BadReleaseException();
        holder = null;
        notifyAll();
    }

    /* Attempt to acquire the lock; return true if successful or
       false otherwise. Does not block. */
    public synchronized boolean attempt() {
        if (holder == null) {
            holder = Thread.currentThread();
            return true;
        }
        else
            return false;
    }
}
```

4. Programming with Threads (26 points).

Classes that implement the following interface can be used by other classes to execute code on their behalf:

```
public interface Executor {
    public void execute(Runnable command);
}
```

Write two implementations of this interface.

- (a) (8 points) Use the thread-per-message model, in which an executor forks off a separate thread for executing the command. Do not worry about bounding the number of threads.

Answer:

```
public class ThreadedExecutor implements Executor {
    public void execute(Runnable command) {
        (new Thread (command)).start();
    }
}
```

- (b) (18 points) Use the bounded thread pool model, in which an executor queues the job to execute, and one or more threads drain the queue to run the jobs. This is similar to the idea of EventThreads in project 4. Implement the PooledExecutor class and a PooledWorkerThread class, using the following Queue interface:

```
public interface Queue {
    void enqueue(Object o) throws QueueFullException;
    Object dequeue() throws QueueEmptyException;
    int currentSize();
    int maxSize();
}
```

You can assume that all of the methods of an implementation of Queue will be synchronized methods. More importantly, notice that the queue will not block when consistency conditions are met, but rather will fail by throwing an exception. In particular, if you try to enqueue on a full queue, it will throw an exception; likewise if you try to dequeue from an empty queue, it will throw an exception.

Your PooledExecutor class and PooledWorkerThread class will be responsible for handling queue failures and waiting until conditions are acceptable before proceeding. In particular, notice that neither PooledExecutor nor PooledWorkerThread throw QueueFullException or QueueEmptyException. As such, a call to execute() should block while the PooledExecutor's queue is full, and a similar situation will occur for PooledWorkerThread when the queue is empty.

Answer:

```
public class PooledExecutor implements Executor {

    private Queue queue;

    PooledExecutor(Queue queue, int numPooledThreads) {
        this.queue = queue;
        for (int i = 0; i < numPooledThreads ; i++) {
            (new PooledWorkerThread (queue)).start();
        }
    }

    public void execute(Runnable command) {
        while (true) {
            synchronized(queue) {
                try {
                    queue.enqueue(command);
                    queue.notifyAll();
                    break;
                }
                catch (QueueFullException e) {
                    try { queue.wait();}
                    catch (InterruptedException e1) {}
                }
            }
        }
    }
}
```

```
public class PooledWorkerThread extends Thread {
    private Queue queue;

    public PooledWorkerThread(Queue queue) { this.queue = queue; }

    public void run() {
        while (true) {
            synchronized (queue) {
                try {
                    ((Runnable) queue.dequeue()).run();
                    queue.notifyAll();
                }
                catch (QueueEmptyException e) {
                    try { queue.wait(); }
                    catch (InterruptedException e1) {}
                }
            }
        }
    }
}
```