

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	

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: }
```

- Example 2:

```
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: }
```

- Example 3:

```
1: class Ex3 {
2:     private final int x;
3:     private final int y;
4:     public Ex3(int init_x, int init_y)
5:     { new Thread()
6:       { public void run()
7:         { System.out.println("x=" + x + " y=" + y);
8:         }
9:       }.start();
10:     x = init_x;
11:     y = init_y;
12:   }
13: }
```

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:

- (a) 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 be deadlocked.
- (b) 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.
- (c) 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.

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.

```
public class ThreadedExecutor implements Executor {
```

```
}
```

- (b) (17 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.

```
public class PooledExecutor implements Executor {

    public PooledExecutor(Queue queue, int numPooledThreads) {

    }

    public void execute(Runnable command) {

    }

}

public class PooledWorkerThread extends Thread {
    private Queue queue;
    public PooledWorkerThread(Queue queue) { this.queue = queue; }
    public void run() {

    }

}
```