Midterm #1

CMSC 433: Programming Language Technologies and Paradigms

October 14, 2013

Name ____

Instructions

Do not start until told to do so!

- This exam has 10 double-sided pages (including this one); make sure you have them all
- You have 75 minutes to complete the exam
- The exam is worth 100 points. Allocate your time wisely: some hard questions are worth only a few points, and some easy questions are worth a lot of points.
- If you have a question, please raise your hand and wait for the instructor.
- In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Question	Points	Score
1	Short Answer	16	
2	Deadlock and Visibility	16	
3	Concurrency Errors	25	
4	Concurrent Program Executions	18	
5	Concurrent Programming	25	
	Total	100	

1. Short answer (16 points total, 4 points each)

   (a) Why are locks called a *concurrency control* mechanism?

   Answer:

   Only one thread can acquire a lock at a time, so they can be used to prevent threads from running, thus "controlling concurrency"

   (b) What does it mean for a lock to be *reentrant*?

   Answer:

   When the same thread can acquire a lock it already holds

   (c) What are the conditions required for an object to be considered *immutable*?

   Answer:

   Its fields must be final, the content of objects referenced in those fields must not change once it is initialized, and the `this` *pointer should not be leaked when running the constructor*

   (d) Name one advantage and one disadvantage of a *copy-on-write* data structure.

   Answer:

   Advantage: no synchronization required on reads. Disadvantage: each write requires making a copy, which can be expensive, rather than an update in place.

2. (Visibility and Deadlock, 16 points, 4 points each)

   (a) Consider the following execution trace:

   `write(t1,x,5); spawn(t1,t2); read(t2,x,5); write(t1,x,6)`

   Does this trace exhibit a data race? If so, circle the operations involved.

   Answer:

   Yes; the last two operations are the ones involved. The first write is not *involved, since it is certain to happen before the first read.*

   (b) Considering the trace above again: Suppose the next operation in the trace is a read by thread `t2` from variable `x`; what value (or values) could be legally read?

   Answer:

   Either 5 or 6, since 5 is the last value to be written that happens before the read, but 6 was written earlier in the trace, but was not ordered properly with the read.

   (c) Name the four necessary and sufficient conditions for deadlock

   Answer:

   Mutual exclusion, no preemption, hold and wait, circular waiting.

   (d) If threads in a program only ever hold one lock at a time, which one of the four conditions necessary for deadlock is not being met?

   Answer:

   Hold and wait. (Note that circular waiting *is not the best answer because it presupposes hold and wait.)*

3. Concurrency errors (25 points, 5 points each)

This problem shows you a series of Java classes, with each followed by a description of threads that use those classes: the *main* thread runs first; when its code is finished it spawns threads T_1 and T_2, which may run in parallel, and may access any variables created by the main thread.

Indicate whether the program could exhibit *deadlock*, a *data race*, an *atomicity violation*, or that it is *correct*.

Assume that all defined methods are meant to be atomic. If the program could exhibit multiple problems, indicate each one. Highlight *exactly* the lines of code involved in any data race or deadlock you find; for any atomicity violation, give a brief explanation of what is going on.

(a)
```
public class Counter {
  private Integer value;
  public Counter(int v) { value = v; }
  public void twiceBump(int amt) {
    synchronized (value) {
      value += amt;
      value += amt;
    }
  }
}
```

main: `Counter ctr = new Counter(0);`
T_1: `ctr.twiceBump(4);`
T_2: `ctr.twiceBump(4);`

Answer:

This is both a data race and an atomicity violation because we are synchronizing on `value`*, but this field changes. Thus T_1 could acquire the lock on the original value, modify this value on line 5, and then have T_2 acquire the lock on the modified value, leading to both threads executing* `twiceBump` *together.*

(b)
```
public class Counter {
  private volatile int value;
  public Counter(int v) { value = v; }
  public void twiceBump(int amt) {
    value += amt;
    value += amt;
  }
}
```

main: `Counter ctr = new Counter(0);`
T_1: `ctr.twiceBump(4);`
T_2: `ctr.twiceBump(4);`

Answer:

There is a straight atomicity violation here since no synchronization is used. There is no data race, since the variable is `volatile`*, and therefore the reads and writes made by the two threads are happens-before ordered.*

(c)
```
public class Counter {
  private final Integer value;
  public Counter(int v) { value = v; }
  public Counter twiceBump(int amt) {
    int v = value + 2*amt;
    return new Counter(v);
  }
}
```

main: `Counter ctr = new Counter(0);`
T_1: `ctr.twiceBump(4);`
T_2: `ctr.twiceBump(4);`

Answer:

It's correct! The counter is immutable.

(d)
```
public class Compound {
  public static <K,V> boolean putIfBothAbsent(
    ConcurrentHashMap<K,V> map, K key1, K key2, V val1, V val2)
  {
    synchronized (map) {
      if (map.get(key1) == null && map.get(key2) == null) {
        map.put(key1,val1);
        map.put(key2,val2);
        return true;
      }
    }
    return false;
  }
}
```

main: `ConcurrentHashMap<String,String> m = new ConcurrentHashMap();`
T_1: `m.put("hello","my friend");`
T_2: `putIfBothAbsent(m,"hello","there","see you","later");`

Answer:

This code has an atomicity violation: the `synchronized` *statement does not stop* T_1 *and* T_2 *from running concurrently. The mistake is an attempt to use client-side locking on a* `ConcurrentHashMap`*, which does not use its intrinsic lock.*

(e)
```
public class BankAccount {
  private int balance = 0;
  private Object balanceLock = new Object();
  private int numOps = 0;
  private Object opLock = new Object();

  public boolean withdraw(int amt) {
    synchronized (balanceLock) {
      if (amt > balance) { return false; }
      balance -= amt;
      synchronized (opLock) {
        numOps++;
      }
    }
    return true;
  }

  public void deposit(int amt) {
    synchronized (opLock) {
      numOps++;
      synchronized (balanceLock) {
        balance += amt;
      }
    }
  }
}
```

main: `b = new BankAccount(); b.deposit(100);`
T_1: `b.withdraw(50);`
T_2: `b.deposit(50);`

Answer:

There is a possible deadlock (all four `synchronized` *statements are involved), due to the locks being ordered differently in* `deposit` *and* `withdraw`

4. (Concurrent executions, 18 points total, 6 points each)

   The following programs are correct. **Indicate what each program print when run. If there are multiple possible outcomes, list all of them.** (The format of the program is the same as the previous question: the main thread runs before the other threads, which may run in parallel, and may access variables created in the main thread.)

   (a)
```
public class Coord {
  private final BlockingQueue<String> q = new LinkedBlockingQueue(1);
  public void push(String v) {
    q.add(v); System.out.println("added "+v);
  }
  public String pop() throws InterruptedException {
    return q.take();
  }
}
```

   main: `q = new Coord();`
   T_1: `s = q.pop(); q.push(s+" again");`
   T_2: `q.push("going");`
   T_3: `q.pop();`

   Answer:

   Can print either

   i. *"added going" or*
   ii. *"added going" followed by "added going again" or*
   iii. *"added going again" followed by "added going"*

   (b)
```
public class CHMap {
  public static void go(ConcurrentHashMap<String,String> map) {
    if (map.get("hello") == null) {
      map.put("snuck","in");
      System.out.println(map.size());
    }
  }
}
```

   main: `map = new ConcurrentHashMap();`
   T_1: `go(map);`
   T_2: `map.put("hello","there");`

   Answer:

   Three possibilities: prints either 1 or 2 or nothing at all

(c)
```
public class Await {
  private CyclicBarrier barrier = new CyclicBarrier(3);
  private int val = 0;
  public synchronized void inc() { val = val + 1; }
  public void go() throws Exception {
    inc();
    barrier.await();
    System.out.println("val = "+val);
  }
}
```

main: `a = new Await();`
T_1: `a.go();`
T_2: `a.go();`
T_3: `a.go();`

Answer:

Always prints "val = 3" three times

5. (Concurrent programming, 25 points)

   Implement the `DoubleStack` class in a thread-safe manner. This class has the following signature:

```
public class DoubleStack {
  public void pushLeft(int elem);
  public void swapStacks();
  public int popRight() throws NoSuchElementException;
  public List<Integer> clearRight();
  public int maxLeft() throws NoSuchElementException;
}
```

   A `DoubleStack` object maintains two stacks, a left one and a right one. When it is first created, both are empty. The `pushLeft` left pushes an element onto the left stack. The `swapStacks` method swaps the left and right stack. The `popRight` pops an element off of the right stack; if the stack is empty, it throws an `NoSuchElementException`. The `clearRight` method clears the right stack, returning all of its elements as a list, where the first (at index 0) is the most recently pushed. Finally, the `maxLeft` method returns the maximum integer in the left stack, and throws `NoSuchElementException` if that stack is empty.

   Your implementation should be as efficient as possible, while still being thread-safe.

   Write your code on the next page.

   Here we list some method names from the `LinkedList` class, for your reference; you are not obligated to use them.

```
class LinkedList<T> implements List<T> {
  public LinkedList<T>(); // makes linked list with elements of type T
  public boolean add(T x); // adds x to the end of the list; returns true
  public Object clone(); // returns a shallow copy of this list
  public void clear(); // removes all elements from the list
  public T get(int n); // returns the element at index n, else null
  public boolean isEmpty(); // returns whether the list is empty
  public ListIterator<T> listIterator(); // returns a list iterator
  public T pop() throws NoSuchElementException;
    // removes the element from the front of the list and returns it
    // throws exception if the list is empty
  public void push(T x); // adds x to the front of the list
  public void remove(int n) throws IndexOutOfBoundsException;
    // removes the element at index n, or an exception if the=
    // index is less than 0 or greater than the size() - 1
  public int size(); // returns the number of elements in the list
  ...
}
```

(Put your answer here) **Answer:**

Here's an answer using two locks. If you use one lock, you can just synchronize each method.

```
import java.util.LinkedList;
import java.util.List;
import java.util.NoSuchElementException;

public class DoubleStack {
  private LinkedList<Integer> leftStack = new LinkedList<Integer>();
  private LinkedList<Integer> rightStack = new LinkedList<Integer>();
  private final Object leftLock = new Object();
  private final Object rightLock = new Object();

  public void pushLeft(int elem) {
    synchronized (leftLock) {
      leftStack.push(elem);
    }
  }
  public void swapStacks() {
    synchronized (leftLock) {
      synchronized (rightLock) {
        LinkedList<Integer> tmp = leftStack;
        leftStack = rightStack;
        rightStack = tmp;
      }
    }
  }
  public int popRight() throws NoSuchElementException {
    synchronized (rightLock) { return rightStack.pop(); }
  }
  public List<Integer> clearRight() {
    synchronized (rightLock) {
      LinkedList<Integer> tmp = rightStack;
      rightStack = new LinkedList<Integer>();
      return tmp;
    }
  }
  public int maxLeft() throws NoSuchElementException {
    int max;
    synchronized (leftLock) {
      if (leftStack.isEmpty()) throw new NoSuchElementException();
      max = leftStack.get(0);
      for (Integer elem: leftStack) {
        if (elem > max) {
          max = elem;
        }
      }
      return max;
} } }
```