

Final Exam - Answer Key

CMSC 433: Programming Language Technologies and Paradigms

December 16, 2010

Name _____

Instructions

Important

- **Two times** you may write **Punt** on any part of a question and receive five points. Thus if you punt a 10-point question, you will only receive 5 points for it.
- *Do not start until told to do so!*

Routine

- This exam has 16 pages (including this one); make sure you have them all.
- You have 120 minutes to complete the exam
- The exam is worth 110 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.
- You may use the back of the exam sheets if you need extra space.
- **Write neatly and clearly indicate your answers.**

	Question	Points	Score
	Short Answer	30	
	Java Concurrency	30	
	Erlang	25	
	Erlang Servers & DSU	15	
	Parallelization	10	
	Total	110	

Enjoy your break!

1. (Short Answer, 30 points)

- (a) (5 points) When writing a Java program you have the option of using one of two styles of thread-safe collection: a *concurrent collection*, like `ConcurrentHashMap`, or a *synchronized collection*, like one returned from `Collections.synchronizedMap`. Give a benefit of each.

Answer:

A concurrent collection allows multiple threads to be reading/modifying it in parallel, whereas a synchronized collection does not. On the other hand, it is hard to make compound operations atomic (e.g., of the flavor read-modify-write), since you cannot use client-side locking.

- (b) (5 points) Give one reason why it is hard to test multithreaded programs.

Answer:

The main problem is nondeterminism from scheduling. For example, one run of the program might reveal a data race, but subsequent runs do not, making the bug hard to track down.

- (c) (5 points) What problem does **volatile** solve?

Answer:

*It solves the visibility problem. The Java Memory Model does not ensure sequential consistency, so an unsynchronized write to a field by one thread may not be seen by a subsequent read from that field from another thread. Using **volatile** restores sequential consistency for the given field.*

- (d) (5 points) MapReduce is inspired by functional programming, and the Map and Reduce are supposed to be “functional” or “not have side effects,” such as changing **static** fields or writing output other than via **Context**. Give an advantage of this restriction.

Answer:

The results produced by a mapper or reducer are deterministic: they are purely a function of the inputs. Therefore if a node running one of these tasks crashes it is easy to restart the computation on a different node, somewhere else.

- (e) (5 points) Is this class thread-safe? (Explain your answer for partial credit.)

```
1 public class Utilities {  
2     public final int extra;  
3     public Utilities (int extra) { this.extra = extra; }  
4     public int sum(int x, int y) { return x+y+extra; }  
5     public int product(int x, int y) { return x * y * extra; }  
6 }
```

Answer:

Yes, because it is immutable.

- (f) (5 points) What is an advantage of using Executors instead of creating a new Thread for each task you want to perform?

Answer:

It allows you to decouple the specification of tasks from the specification of resources to execute those tasks. Creating a thread each time allots one thread per task, all running in parallel. If you have many tasks and few processors, this could be very expensive (lots of context switching). Using an Executor, e.g., a thread pool, allows you to specify only as many threads as you have processors, and each thread can execute a task one at a time, reducing overhead.

2. (Java Concurrency, 30 points)

Each of the next few pages contains a small program with at least two classes, one of which is called `TestCase`. Consider what would happen when running `TestCase.main()`. If the program would terminate normally and always print the same answer, indicate that answer. Otherwise, indicate one or more of the following the things the program will do: (a) exhibit a data race, (b) exhibit an atomicity violation, (c) exhibit a deadlock, (d) run forever, (e) print different things on different runs.

For full credit, briefly explain your answer (1-2 sentences). (You may want to use the back of the page.) Each problem is worth 5 points.

Be careful: I am only interested in what will happen for executions of the given `TestCase.main`, not hypothetical ways in which the classes could be used.

(a)

```
1  public class Service implements Runnable {
2      public static int liveThreads = 0;
3      public static boolean done() { return liveThreads == 0; }
4      public Service() { liveThreads++; }
5      public void run() { liveThreads--; }
6  }
7
8  public class TestCase {
9      public static void main(String args[]) {
10         for (int i=0;i<10;i++) {
11             new Thread(new Service()).start();
12         }
13         while (!Service.done()) {} // do nothing
14         System.out.println("done!");
15     }
16 }
```

Answer:

Atomicity violation on `liveThreads` due to read-modify-write data races (in constructor and in `run`); as a result one of the updates might get lost, or a stale value may be read by `main`, so the program may not terminate.

(b)

```
1  public class Logger {
2      private ArrayList<String> records = new ArrayList<String>();
3      // log an event
4      public void add(String event) {
5          synchronized (records) {
6              records.add(event);
7          }
8      }
9      // return (and remove) last event logged
10     public String last () {
11         synchronized (records) {
12             int sz = records.size ();
13             if (sz > 0) return records.remove(sz-1);
14             else return null;
15         }
16     }
17 }
18
19 public class TestCase {
20     public static void main(String args[]) {
21         final Logger log = new Logger();
22         log.add("Hello");
23         // each thread tries to remove and print last event
24         for (int i=0;i<2;i++) {
25             new Thread() {
26                 public void run() {
27                     String s = log.last ();
28                     if (s != null)
29                         System.out.println(s);
30                 }
31             }.start ();
32         }
33     }
34 }
```

Answer:

Deterministic outcome: prints "Hello" (not always printed from the same thread, but that's immaterial)

(c)

```
1  public class NameServer {
2      // location map: person (String) maps to their current location (String)
3      public final ConcurrentHashMap<String,String> locations =
4          new ConcurrentHashMap<String,String>();
5      // Have two people trade places
6      public void tradePlaces(String name1, String name2) {
7          String loc1 = locations.get(name1);
8          String loc2 = locations.get(name2);
9          locations.put(name1,loc2);
10         locations.put(name2,loc1);
11     }
12 }
13
14 public class TestCase {
15     public static void main(String args[]) throws InterruptedException {
16         // sets initial locations
17         final NameServer n = new NameServer();
18         n.locations.put("john", "home");
19         n.locations.put("bill", "work");
20         // two threads that trade john & bill's places
21         Thread t[] = new Thread[2];
22         for (int i=0; i<2; i++) {
23             t[i] = new Thread() {
24                 public void run() {
25                     n.tradePlaces("john", "bill");
26                 }
27             };
28             t[i].start();
29         }
30         t[0].join();
31         t[1].join();
32         System.out.println("John is at "+n.locations.get("john"));
33         System.out.println("Bill is at "+n.locations.get("bill"));
34     }
35 }
```

Answer:

There is an atomicity violation: `tradePlaces` will not be atomic. As such, the program might print "home" then "work" or it might print "work" then "home" if it context-switches at line 8, or even both "home" or both "work" if it context switches at line 9. There is no data race on `locations`, because it always points to the same `ConcurrentHashMap` which is a thread-safe class.

(d)

```
1  public class Node {
2      private boolean haveToken; // whether this Node has a token
3      private int idx;           // Node's id
4      public Node(int id, boolean tok) {
5          idx = id; haveToken = tok;
6      }
7      // hand off token from this Node to target Node
8      public synchronized void pass(Node target) {
9          if (haveToken) {
10             synchronized(target) {
11                 haveToken = false;
12                 target.haveToken = true;
13                 System.out.println(idx+" has passed the token to "+target.idx);
14             }
15         }
16     }
17 }
18
19 public class TestCase {
20     public static void main(String args[]) {
21         final Random r = new Random();
22         final Node[] nodes = new Node[10];
23         for (int i=0; i<10; i++) {
24             nodes[i] = new Node(i, i % 2 == 0); // 10 nodes; half have token
25         }
26         // threads that pass tokens between nodes
27         for (int i=0; i<10; i++) {
28             final int j = i;
29             new Thread() {
30                 public void run() {
31                     // pass token to a random node (idx 0..9)
32                     nodes[j].pass(nodes[r.nextInt (10)]);
33                 }
34             }.start ();
35         }
36     }
37 }
```

Answer:

The program may deadlock, since two nodes could attempt to pass a token to each other and be waiting on the other's lock. If the program does not deadlock, its output is nondeterministic, since we are passing tokens between

random nodes.

(e)

```
1 public class Last {
2     private final AtomicInteger value = new AtomicInteger(0);
3     public void bump(int incr) {
4         while (true) {
5             int oldv = value.get();
6             int newv = oldv + incr;
7             if (value.compareAndSet(oldv, newv))
8                 return;
9         }
10    }
11    public int getLast() { return value.get(); }
12 }
13
14 public static class TestCase {
15     public static void main(String args[]) throws InterruptedException {
16         final Last x = new Last();
17         Thread t[] = new Thread[2];
18         for (int i = 0; i < 2; i++) {
19             t[i] = new Thread() {
20                 public void run() {
21                     x.bump(2); x.bump(2);
22                 }
23             };
24             t[i].start();
25         }
26         t[0].join();
27         t[1].join();
28         System.out.println(x.getLast());
29     }
30 }
```

Answer:

Deterministic: prints 8. The use of `compareAndSet` ensures that calls to `bump` are atomic. The program will terminate because each time around the loop, the call to `get` will retrieve the most recently stored value. Since each thread only calls `bump` twice, the `compareAndSet` is guaranteed to succeed eventually.

(f)

```
1  public class Counter {
2      private int value = 0;
3      public int get() {
4          return value;
5      }
6      public synchronized void incr() {
7          value++;
8      }
9  }
10
11 public class TestCase {
12     public static void main(String args[]) {
13         final Counter counter = new Counter();
14         for (int i = 0; i < 2; i++) {
15             new Thread() {
16                 public void run() {
17                     counter.incr ();
18                     System.out.println ("counter_value_" + counter.get());
19                 }
20             }.start ();
21         }
22     }
23 }
```

Answer:

*Nondeterministic outcome (could print 1 and 2, or both 2), due to a data race (on **get**), which means (due to the visibility rules) that one thread might not see the value previously written by the other thread, e.g., if there was a context switch at line 17. Note that no thread's call to **get** should ever see 0 since the thread's own call to **incr** always happens before the **get** call.*

3. (Erlang, 25 points)

Look at each of the Erlang programs below, and explain what happens when you call `test()`. If it returns, indicate the value returned; if it fails with some exception, indicate the exception; or if it does not return, say so. You may assume all of the programs compile. Each problem is worth 5 points.

You *do not* have to explain your answers, but you may do so for possible partial credit.

```
(a) go({Z,hello}) -> Z;
    go({{X,Y},bye}) -> Y;
    go({-, -}) -> 0.
    test() ->
        go({{1,2},hello}).
```

Answer:

`{ 1,2 }`

```
(b) go(L) -> lists:map(fun (X) -> X*X end,L).
    test() ->
        L = go([1,2,3]),
        L = go([3,4]).
```

Answer:

Fails with bad pattern match.

```
(c) test() ->
    Pid = self (),
    L = [1,2,3],
    lists:foreach(fun(X) -> Pid ! 2-X end,L),
    lists:map(fun(_) -> receive X -> X end end,L).
```

Answer:

`[1,0,- 1]`

```
(d) test () ->
    Pid = self (),
    spawn(fun () -> Pid ! 100, Pid ! 25 end),
    receive
        {X,Y} -> X;
        100 -> 10;
        Z -> Z
    end.
```

Answer:

10

```
(e) parent_loop(Acc) ->
    receive
        {ok,X} -> parent_loop([X|Acc]);
        {'EXIT', _Pid, _Why} -> Acc
    end.
```

```
test () ->
    Pid = self (),
    process_flag( trap_exit ,true),
    CPid = spawn(fun () -> timer:sleep(100),
        Pid ! {ok,1},
        Pid ! {ok,"hello"},
        ok end),

    link(CPid),
    parent_loop([]).
```

Answer:

["hello",1]

4. (Erlang Servers and DSU, 15 points)

The code on the next page is in the style of the sequence and semaphore examples we saw in class: there is a loop that maintains some state (in this case the integer `CurVal`) and the loop code receives messages and acts on them. In this case, rather than hardcoding the functionality of the loop, we initialize the server to use a function `F` of our choosing; each time the message `{From,Id,doit}` is received by the server, it invokes function `F` on the current state and sends back the result.

Now suppose we wish to be able to update the loop's function `F` on the fly. Implement the `code_swap` function, which takes the `Pid` of the loop process as its first argument, and a new function that the loop should use as its second argument. After this function returns, subsequent calls to `doit` should cause the looping process to use the new function. The testcase at the bottom illustrates how this should work.

(a) (5 points) *Do the next part of this question first, then come back to this part.*

As discussed in class, there are effective restrictions on how you can change your program based on what it is doing when the update takes place.

Illustrate this fact by extending the test case on the next page: suppose we insert the following code sequence just after the line containing `12 = doit(P)`, in the test case:

```
code_swap(...),  
doit(P)
```

What could you put into ... so that the call to `doit` will cause the looping process to crash?

Answer:

It could be any function that will do the wrong thing if given a single integer argument. For example:

`code_swap(P,fun (X,Y) -> X + Y end), doit(P)`

In this case, this is a two-argument function, so there will be a "bad argument" failure on the call to `doit`.

Next page $\Rightarrow$...

(b) (10 points) Fill in the missing bits below:

```
make(F,N) -> spawn(fun() -> loop(F,N) end).
```

```
loop(F,CurVal) ->
```

```
  receive
```

```
    {From, Id, doit} ->
```

```
      NewVal = F(CurVal),
```

```
      From ! {Id, NewVal},
```

```
      loop(F,NewVal);
```

```
    %% FILL IN: ANSWER:
```

```
    {code_swap, NewF} ->
```

```
      loop(NewF,CurVal)
```

```
  end.
```

```
doit(P) ->
```

```
  Id = make_ref(),
```

```
  P ! {self (), Id, doit},
```

```
  receive {Id, CurVal} -> CurVal end.
```

```
code_swap(P,NewF) ->
```

```
  %% FILL IN: ANSWER:
```

```
  P ! {code_swap, NewF}.
```

```
% test case (should produce no match failures)
```

```
test () ->
```

```
  P = make(fun (X) -> X + 1 end,5), % init with inc. func, value 5
```

```
  6 = doit (P), % increments P, value now 6
```

```
  7 = doit (P), % increments P, value now 7
```

```
  code_swap(P,fun (X) -> X - 1 end), % change to use the dec. function
```

```
  6 = doit (P), % decrements P, value now 6
```

```
  code_swap(P,fun (X) -> X * 2 end), % change to use doubling function
```

```
  12 = doit (P), % doubles P, value now 12
```

```
  ok.
```

5. (Parallelization, 10 points)

The method `matchParentheses` attempts to match parentheses in a given array. The result of the matching is given as a pair, which indicates how many open and closed parentheses are unmatched. We represent, in the array, open-paren as 0, and closed-paren as 1. As such, suppose array is `[0,0,0,0,1,1]`, representing `((((()))`, then `matchParentheses(array,0,6)` would return a `Result` with `closed=0,open=2`. If array represented `))()()` then the `Result` of `matchParentheses(array,0,6)` would contain `closed=2,open=1`.

```
public class Result {
    public final int open, closed;
    public Result(int open, int closed) {
        this.open = open;
        this.closed = closed;
    }
}

...
public static Result matchParentheses(int[] array, int start, int end) {
    int open = 0, closed = 0;
    for (int i = start; i < end; i++) {
        // the details of how this is implemented don't matter ...
    }
    return new Result(open,closed);
}
```

Sketch how you would implement a method `matchParenParallel(int[] array)` that does parenthesis matching to produce the `Result` of an entire array in parallel, using the above method as a subroutine. You can provide pseudocode, text, and/or pictures to explain. Say what implementation framework you would to implement this, e.g., Java thread pools and tasks, Java fork-join, Hadoop, etc. and make sure how you'd use this framework is clear in your explanation.

Answer:

You can use fork-join. Basically you will keep dividing the array in half, processing it in separate tasks until it falls below a certain minimum size. When the array is below the minimum size, you call `matchParentheses` on it to get the answer, returned as a `Result`. Otherwise, you combine the two `Results` of the two children, as follows. Let $middle = left.result.open - right.result.closed$. Then, if $middle \geq 0$ then the final `Result`'s `open` is $left.result.open + middle$ and the `closed` is just the $right.result.closed$. Otherwise, the final `open` is $left.result.open$ and its `closed` is $right.result.closed + middle$.

You could also implement this by dividing the array into N parts, where N is the number of processors or something larger if the array is too small (so chunks stay above a minimum size). Then you would call `matchParentheses` on each of these in parallel, and then combine the results, left to right.

Using map-reduce is more clumsy, since mapping tasks emit using key-value pairs. Probably you would have a single key for the output, and so a single reduce task would handle the list of results associated with that key. This list would have to include an identifier to associate each result with its position in the original array.