Final Exam - Answer Key

CMSC 433: Programming Language Technologies and Paradigms

December 20, 2013

Name _____

Instructions

- This exam has 15 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.
- **Write neatly** and **clearly indicate your answers**.

	Question	Points	Score
	Definitions	20	
	Distributed Computing	20	
	Java Concurrency	30	
	Erlang Execution	20	
	Erlang Coding	10	
	Parallelization	10	
	Total	110	

Enjoy your break!

1. Definitions (20 points total, 2 points each).

   Next to each term on the left, write the letter on the right that corresponds to the definition of the term, or is a true statement about it.

	(a) A technique for ensuring the atomicity of multiple method calls
____ Marshaling	(b) Performing garbage collection using multiple parallel threads
	(c) Performing garbage collection in parallel with the mutator
____ Parallel GC	(d) A property of the Java Memory Model (JMM)
	(e) A class that satisfies its specification when used by multiple threads
____ Sequential consistency	(f) An example of which is calling BlockingQueue.take
	(g) A lock that can be acquired by multiple threads
____ Thread-safe class	(h) A property of executions in which the effects of separate threads' operations can be viewed as happening in a total order
____ Client-side locking	(i) Erlang uses this style of typing
____ State-dependent action	(j) A lock that is particularly important when synchronized methods call other synchronized methods
	(k) Predicts a task's possible speedup based on the number of processors and the amount of strictly-serial work
____ Cast (as in Erlang)	(l) An action involving an object implementing a finite state machine
____ Dynamic typing	(m) A message sent to a server that expects a response
	(n) Predicts how many threads you should create based on the number of processors and the ratio of wait time to compute time
____ Reentrant lock	(o) A message sent to a server that needs no response
____ Amdahl's Law	(p) A class with no data races
	(q) Java uses this style of typing
	(r) The process of preparing a remote method invocation

Answer:

(r), (b), (h), (e), (a), (f), (o), (i), (j), (k)

2. (Distributed computing, 20 points)

(a) (Scala Actors, 4 points) You can send a message to a Scala actor `a` using the code `a ! msg`. Scala also allows you to send messages via the syntax `a !! msg` and the syntax `a !? msg`. How will these two invocations differ from the first?

Answer:

The first sends a message and immediately continues. The second sends a message and returns a future that will contain the response. The third sends a message and waits until the response is provided.

(b) (Java RMI, 4 points) Consider the following two possible definitions of a class MyString:

Implementation (a):

```
1 public class MyString {
2   private final String contents;
3   public String getContents() { return contents; }
4 }
```

Implementation (b):

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3 public interface RemoteString extends Remote {
4   public String getContents() throws RemoteException;
5 }
6 public class MyString implements RemoteString {
7   private final String contents;
8   public String getContents() throws RemoteException { return contents; }
9 }
```

Suppose `x` is a reference to a *remote* object of type `Foo`, which has a method `f` that takes a `MyString` argument, and `y` is a reference to a local `MyString` object. If we invoke `x.f(y)` what will be the difference in the case that we use implementation (a) vs. implementation (b) ?

Answer:

For implementation (a) we will serialize `y` and send it to `x`'s remote location to execute `f`. For implementation (b), we will send a remote reference of `y` to `x`'s remote location, so that when `f` is executed there, accessing `y`'s contents will result in messages back to its home location.

(c) (MapReduce, 12 points) The following Hadoop code is based on the project 5 tutorial:

```
1  public static class TokenizerMapper extends
2       Mapper<Object, Text, IntWritable, IntWritable> {
3    private final static IntWritable one = new IntWritable(1);
4    private IntWritable count = new IntWritable(0);
5
6    public void map(Object file, Text value, Context context)
7         throws IOException, InterruptedException {
8      StringTokenizer itr = new StringTokenizer(value.toString());
9      while (itr.hasMoreTokens()) {
10       int length = itr.nextToken().length();
11       count.set(length);
12       context.write(count, one);
13 } } }
14
15 public static class IntSumReducer extends
16      Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
17   private IntWritable result = new IntWritable();
18
19   public void reduce(IntWritable key, Iterable<IntWritable> values,
20        Context context) throws IOException, InterruptedException {
21     int sum = 0;
22     for (IntWritable val : values) {
23       int v = val.get();
24       sum += v;
25     }
26     result.set(sum);
27     context.write(key, result);
28 } }
```

i. (6 points) What will the keys and values be in the final output file, assuming this code is run on a directory of text files (which consist of, say, the collected works of Shakespeare)? **Answer:**

The output file's keys will be lengths of words found in the input files, and the values will the number of words that appeared having that length.

ii. (6 points) Could IntSumReducer also be used as a combiner? Why or why not? **Answer:**

Yes, because it is simply summing counts, and summing is both associative and commutative

3. (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 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.

   Each problem is worth 6 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.

   Also, note that in several cases we are ignoring that methods could throw InterruptedException, to keep the code shorter.

   (a)

```
1  public class GlobalRef {
2    public int x = 0;
3    public Object y = new Object();
4    public void increment() { synchronized(y) { x = x + 1; } }
5  }
6
7  public class TestCase {
8    public static void main(String args[]) {
9      final GlobalRef r = new GlobalRef();
10     new Thread() { public void run() { r.increment(); } }.start();
11     new Thread() { public void run() { r.increment(); } }.start();
12     Thread.yield();
13     System.out.println(r.x);
14   }
15 }
```

   Answer:

   This program has a data race on `r.x` *between the main thread on line 12 and the accesses on line 4. Due to the visibility rules and the non-determinism of thread scheduling, it could print different things on different runs (in particular, either 0, 1, or 2). Note there is no atomicity violation here, since* increment *is indeed atomic—it is* `r.x` *that could see a stale value.*

(b)

```
1  public class OnePlaceBuffer {
2    private String value = null;
3    public synchronized String get() {
4      while (value == null) wait (); // ignore IE
5      String ret = value;
6      value = null;
7      notify ();
8      return ret;
9    }
10   public synchronized void put(String x) {
11     while (value != null) wait (); // ignore IE
12     value = x;
13     notify ();
14   }
15 }
16
17 public class TestCase {
18   public static void main(String args[]) {
19     final OnePlaceBuffer b = new OnePlaceBuffer();
20     Runnable p = new Runnable() {
21       public void run() { b.put("Hello"); }
22     };
23     new Thread(p).start();
24     new Thread(p).start();
25     System.out.println(b.get ());
26   }
27 }
```

Answer:

This program will always print "hello" (despite using notify *and not* notifyAll *).*

(c)

```
1  public class SillyFact {
2    private ExecutorService exec;
3    public SillyFact(ExecutorService e) { exec = e; }
4    public int nth(int n) {
5      if (n == 0) return 1;
6      else {
7        final int m = n−1;
8        final SillyFact me = this;
9        Callable<Integer> c = new Callable<Integer>() {
10         public Integer call () { return me.nth(m); }
11       };
12       Future<Integer> f = exec.submit(c);
13       return n + f.get(); // ignore IE
14     }
15   }
16 }
17
18 public class TestCase {
19   public static void main(String args[]) {
20     ExecutorService e = Executors.newFixedThreadPool(4);
21     SillyFact f = new SillyFact(e);
22     System.out.println(f.nth(6));
23     e.shutdown();
24   }
25 }
```

Answer:

This program has a thread starvation deadlock because the Executor *we are using has a fixed number of threads (8), and yet the recursive call to* nth *will eventually exceed this number, and thus each thread will be stuck waiting, and never complete.*

(d)

```
1  public class Pair {
2    public final int left, right;
3    public Pair(int x, int y) { left = x; right = y; }
4  }
5  public class ConcPair {
6    private Pair p = new Pair(0,0);
7    public void updateLeft(int x) {
8      int y = p.right;
9      p = new Pair(x,y);
10   }
11   public void updateRight(int x) {
12     int y = p.left;
13     p = new Pair(y,x);
14   }
15   public int getLeft() { return p.left; }
16   public int getRight() { return p.right; }
17 }
18
19 public class TestCase {
20   public static void main(String args[]) throws InterruptedException {
21     final ConcPair m = new ConcPair();
22     Thread t1 = new Thread() { public void run() { m.updateLeft(1); } };
23     Thread t2 = new Thread() { public void run() { m.updateRight(1); } };
24     t1.start(); t2.start();
25     t1.join(); t2.join();
26     System.out.println("("+m.getLeft()+","+m.getRight()+")");
27   }
28 }
```

Answer:

This program has a data race in both updateLeft *and* updateRight *that violate atomicity, and could thus produce multiple outputs. For example it could produce (1,0) and (0,1).*

(e) Note: Here, OptWhiteList is a different implementation of ConcPair from the previous question; the Pair class is the same as before; the TestCase class is the same, but refers to OptWhiteList

```
1  public class OptWhiteList {
2    private AtomicReference<Pair> p;
3    public OptWhiteList() { p = new AtomicReference<Pair>(new Pair(0,0)); }
4    private void update(boolean doLeft, int x) {
5      while (true) {
6        Pair q, old = p.get();
7        if (doLeft) q = new Pair(x,old.right);
8        else        q = new Pair(old.left,x);
9        if (p.compareAndSet(old,q)) break;
10     }
11   }
12   public void updateLeft(int x) { update(true,x); }
13   public void updateRight(int x) { update(false,x); }
14   public int getLeft() { return p.get().left; }
15   public int getRight() { return p.get().right; }
16 }
17
18 public class TestCase {
19   public static void main(String args[]) throws InterruptedException {
20     final OptWhiteList m = new OptWhiteList();
21     Thread t1 = new Thread() { public void run() { m.updateLeft(1); } };
22     Thread t2 = new Thread() { public void run() { m.updateRight(1); } };
23     t1.start(); t2.start();
24     t1.join(); t2.join();
25     System.out.println("("+m.getLeft()+","+m.getRight()+")");
26   }
27 }
```

Answer:

This program will terminate correctly and print the value "(1,1)".

4. (Erlang Execution, 20 points)

   Look at each of the Erlang programs below. Along with each program, we will provide a call; say what the call will do. If it returns, indicate the value returned; if it fails with some exception, indicate the exception; or if it does not return, say so. Explain your reasoning if you hope for partial credit. You may assume all of the programs compile. Each question is worth 5 points.

   (a) What will go([1,2,3]) do?

```
go([]) -> 0;
go([H|T]) -> 1+go(T).
```

   Answer:

   return 3

   (b) What will go(grill ,burger) do?

```
machine_map() ->
    dict : from_list ([{ grill , {burger, 500}},
                       { frier , { fries , 250}},
                       {soda_fountain, {coke, 100}}]).
go(Machine,Food) ->
  Dict = machine_map(),
  {Food,Time} = dict:fetch(Machine,Dict),
  Time.
```

   Answer:

   return 500

(c) What will go(goo) do?

```
 init () -> register(s1, spawn(fun () -> receive {msg,Pid} -> Pid ! ok end end)).
go(Msg) ->
     init (),
    s1 ! {Msg,self()},
    receive
        ok -> ok
    end.
```

Answer:

Hangs.

(d) What will go() do?

```
pmap(F,L) ->
  Me = self(),
  Pids =  lists :map(fun (M) ->
             spawn(fun () ->
               Pid = self (),
               R = F(M),
               Me ! {Pid,R} end) end,
            L),
   lists :map(fun (Pid) ->
                receive
                  {Pid,R} -> R
                end
              end, Pids).

go() ->
    pmap(fun ([H|T]) -> H end, [[1,2,3],[3,2,1],[3,2],[1,7,5,4]]).
```

Answer:

*return **[1,3,3,1]***

5. (Erlang coding, 10 points) The following function go() calls a function f for which we have not provided a definition. This function will return a process ID to which go subsequently sends messages. Give an implementation of f so that go() returns correctly with the atom **ok**. (You may define other functions too, if you need them.)

```
go() ->
  P = f (),
  P ! [1, 3, 7],
  receive
    {P,3} -> ok
  end,
  P ! [4, 5],
  receive
    {P,2} -> ok
  end,
  P ! done,
  receive
    finished -> ok
  end,
  ok.
```

Give your definition of f here:

Answer:

Here is one possible answer; there are many.

```
f() ->
  Q = self(),
  P = spawn(fun () ->
        receive
          [_,_,_] -> Q ! {self(),3}
        end,
        receive
          [_,_] -> Q ! {self(),2}
        end,
        receive
          done -> Q ! finished
        end
      end),
    P.
```

6. (Java parallelization, 10 points) A Data object has a Grid as a private field. Data also defines a method heatMap to compute this grid's "heat map", which is itself a Grid. Assume that ArrayGrid is an implementation of the Grid interface backed by an array.

   On the next page, provide the implementation of a new Data class method parHeatMap that produces the same result as heatMap, but does it in parallel.

   Extra credit: Assume that the time to create and run a new task is roughly the same as the time to compute the heat map of 100 grid squares. For extra credit, incorporate this information in your solution.

```
1  public interface Grid {
2    public int getWidth(); // x axis length
3    public int getHeight(); // y axis length
4    public int get(int x, int y); // (0,0) is in the upper left
5    public void set(int x, int y, int v); // sets grid(x,y) to v
6  }
7  public class Data {
8    final private Grid grid;
9    public Data(Grid grid) {
10     this.grid = grid;
11   }
12   private int heat(int x, int y) {
13     int v = grid.get(x,y);
14     for (int i = x-1; i<=x+1; i++) {
15       for (int j = y-1; j<=y+1; j++) {
16         if (i >= 0 && i < grid.getWidth() &&
17             j >= 0 && j < grid.getHeight() &&
18             !(i == x && j == y)) {
19           v = v + (grid.get(i,j) / 2);
20         }
21       }
22     }
23     return v;
24   }
25   public Grid heatMap() {
26     Grid results = new ArrayGrid(grid.getWidth(),grid.getHeight());
27     for (int i = 0; i<grid.getWidth(); i++) {
28       for (int j = 0; j<grid.getHeight(); j++) {
29         results.set(i,j,heat(i,j));
30       }
31     }
32     return results;
33 } }
```

You may find the following API calls useful:

Runtime.getRuntime().availableProcessors()	returns the number of available processors
e.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS)	when e is an ExecutorService, this calls waits until all its tasks terminate.

Also see question 3(c) for some useful API calls.

Answer:

An answer like this one, that takes the minimum work size into account, would net you full points plus extra credit.

```
1  private void localHeatMap(final int startx, final int starty,
2            final int width, final int height, final Grid results,
3            final ExecutorService exec) {
4    if (width * height <= 100*2) {
5      exec.submit(new Runnable() {
6        public void run() {
7          int endx = startx+width;
8          int endy = starty+height;
9          for (int i = startx; i<endx; i++)
10           for (int j = starty; j<endy; j++)
11             results.set(i,j,heat(i,j));
12       }
13     });
14   } else {
15     if (width > height) {
16       int inc = 0, w = width/2;
17       if (w*2 != width) inc = 1;
18         for (int i = 0; i<2; i++)
19           localHeatMap(startx+i*w,starty,w+i*inc,height,results,exec);
20     } else {
21       int inc = 0, h = height/2;
22       if (h*2 != height) inc = 1;
23       for (int i = 0; i<2; i++)
24         localHeatMap(startx,starty+i*h,width,h+i*inc,results,exec);
25     }
26   }
27 }
28 public Grid parHeatMap() {
29   int nproc = Runtime.getRuntime().availableProcessors();
30   ExecutorService exec = Executors.newFixedThreadPool(nproc);
31   Grid results = new ArrayGrid(grid.getWidth(),grid.getHeight());
32   localHeatMap(0,0,grid.getWidth(),grid.getHeight(),results,exec);
33   exec.shutdown();
34   try { exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS); } catch (Exception e) { }
35   return results;
36 }
```

(This page intentionally left blank; feel free to use as scratch paper.)

Answer:

This answer would have netted around 8 points: it is highly inefficient to spawn one thread per square.

```
1  public Grid heatMap() {
2    int nproc = Runtime.getRuntime().availableProcessors();
3    ExecutorService exec = Executors.newFixedThreadPool(nproc+1);
4    final Grid results = new ArrayGrid(grid.getWidth(),grid.getHeight());
5    for (int i = 0; i<grid.getWidth(); i++) {
6      for (int j = 0; j<grid.getHeight(); j++) {
7        final int a = i;
8        final int b = j;
9        exec.submit(new Runnable() {
10         public void run() {
11           results.set(a,b,heat(a,b));
12         }
13       });
14     }
15   }
16   exec.shutdown();
17   exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
18   return results;
19 }
```

To get full credit, you would divide up the grid into roughly equal portions according the number of processors available, while avoiding corner cases of tasks becoming too small.