Midterm Exam #2

CMSC 433
Programming Language Technologies and Paradigms
Spring 2012
April 19, 2012

Guidelines

Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

If you have a question, raise your hand and I will come to you. Note, that I am unlikely to answer general questions however. 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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Short answers (15 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. **Longer responses to these questions will not be read.**

(a) When designing a concurrent algorithm, what does it mean for the solution approach to be Data Parallel? Give an example problem in which a data parallel solution approach is likely to be appropriate.

**Answer:**

*The algorithm runs the same task on different subsets of the input data. Examples include searching and sorting.*

(b) The Executor framework allows a program to manage how and when Threads are created. Give two examples of strategies/policies that can be configured when using the Executor framework.

**Answer:**

*Any two of:*

*How many tasks can execute concurrently?*

*In what order will they run?*

*Can tasks be queued if they can’t run immediately?*

*What happens when executor capacity is exceeded?*

(c) Assume that you start a JVM instance and pass it a value for the java.rmi.server.codebase property. True or false: This value is used by this JVM, j, to locate code for classes that this JVM downloads from the network.

**Answer:**

*False*
CAS (20 Points). The CASStats class receives a stream of Integers from multiple Threads. Integers are received by calling the `insert()` method. At any time from any Thread it can be called on to report the maximum and minimum values it has seen so far by calling the `toString()` method. CASStats should be Thread safe. It should, for example, never fail to record the true minimum and maximum values and it should also ensure that updates to the minimum and maximum values are atomic. Fill in the code below to complete this functionality.

Beside the code below you may only use the following methods:
- From `AtomicReference` – `boolean compareAndSet(V expect, V update)` Atomically sets the value to the given updated value if the current value == the expected value.
- From `Math` –
  - `static int min(int a, int b)` Returns the smaller of two int values.
  - `static int max(int a, int b)` Returns the greater of two int values.

```java
// Helper class
class Bounds {
    private int min, max;
    Bounds(int min, int max) { this.min = min; this.max = max; }
    public int getMin() { return min; }
    public int getMax() { return max; }
    public String toString() { return "('' + min + '','' + max + '')"; }
}

public class CASStats {
    AtomicReference<Bounds> val = new AtomicReference<Bounds>();

    public CASStats() {
        val.set(new Bounds(Integer.MAX_VALUE, Integer.MIN_VALUE));
    }

    public void push (int i) {
        // FILL IN CODE HERE
        Bounds oldVal;
        int oldMin, oldMax;
        do {
            oldVal = val.get();
            oldMin = oldVal.getMin();
            oldMax = oldVal.getMax();
            if (!(i < oldMin || i > oldMax)) {
                break;
            }
        } while (!val.compareAndSet(oldVal,new Bounds(Math.min(oldMin, i), Math.max(oldMax, i))));
    }

    public String toString() {
        return val.get().toString();
    }
}
```
Futures and Executors (15 Points). The ExecutorStats class is supposed to compute the minimum and maximum elements in a list of Integers. The ExecutorStats class uses the Bounds class defined in the previous question. For the specific input used below the main() method should print out, “Starting Stats” and “(0,4)”. However, as written, it currently will not. Explain in detail why it does not print the expected output. What can you do to fix the problem?

```
public class ExecutorStats implements Callable<Bounds> {
    private ExecutorService executor = Executors.newSingleThreadExecutor();
    private List<Integer> data = new ArrayList<Integer>(Arrays.asList(0, 3, 4, 2, 1));

    public Bounds call() throws Exception {
        Future<Integer> maxComputation = executor.submit(new Callable<Integer>() {
            public Integer call() throws Exception { return Collections.max(data); } });

        Future<Integer> minComputation = executor.submit(new Callable<Integer>() {
            public Integer call() throws Exception { return Collections.min(data); } });

        return new Bounds(minComputation.get(), maxComputation.get());
    }

    public static void main(String[] args) throws Exception {
        ExecutorStats stats = new ExecutorStats();
        System.out.println("Starting Stats");
        Future<Bounds> result = stats.executor.submit(stats);
        System.out.println(result.get());
        stats.executor.shutdown();
    }
}
```

Answer:

This code will deadlock before printing “(0,4)” because the executor is a single threaded. To fix this problem, you need to use an Executor with more than 1 worker thread such as that returned by Executors.newCachedThreadPool().
4. Fork-Join (20 points). As in the previous question, the following code is supposed to compute the minimum and maximum elements in a list of Integers. Fill in the code below so that it does so using the Fork-Join framework. Note: The Bounds class is defined in the previous questions. Outside of the Bounds class and the ForkJoinStats class shown below, you may only use the following methods:

- From ForkJoinPool – `<T> T invoke(ForkJoinTask`<T>` task) Performs the given task, returning its result upon completion.
- From ArrayList – `List`<E>` subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

```java
public class ForkJoinStats extends RecursiveTask<Bounds> {
    private static ForkJoinPool pool = new ForkJoinPool();
    private List<Integer> data;
    private int mThreshold = /* threshold */;

    ForkJoinStats(List<Integer> data) { this.data = data; }

    protected Bounds compute() {
        if (data.size() < mThreshold) {
            // Collections.min(data) and Collections.max(data) return
            // the minimum and maximum elements in data, respectively
            return new Bounds(Collections.min(data), Collections.max(data));
        } else {

            // FILL IN CODE HERE
            int mid = data.size() / 2;
            Bounds leftBounds = pool.invoke(new ForkJoinStats(data.subList(0, mid)));
            Bounds rightBounds = pool.invoke(new ForkJoinStats(data.subList(mid, data.size())));
            return new Bounds(
                    Math.min(leftBounds.getMin(), rightBounds.getMin()),
                    Math.max(leftBounds.getMax(), rightBounds.getMax()));
        }
    }

    public static void main(String[] args) throws Exception {
        ForkJoinStats stats = new ForkJoinStats(/* pass in data here */);
        System.out.println("Starting Stats");
        System.out.println(pool.invoke(statComputation));
    }
}
```

5. Nonblocking Data Structures (30 Points). The code below implements a nonblocking queue. Threads can safely use the queue without holding a Java lock. Starting with an empty queue, show a trace that involves 2 threads and in which one thread’s put() operation interferes with the other thread’s take() operation. Your trace should be given in table form on the following page. Each row of the table represents one time step in the trace. At each time step indicate the line number that is currently being executed by each thread. If a thread gets blocked or takes more than one time step to execute a single line, repeat the line number over multiple rows.

```java
public class ConcurrentQueue<E> {
    private static class Node<E> {
        final E item;
        final AtomicReference<Node<E>> next;
        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }
    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail = new AtomicReference<Node<E>>(dummy);
    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {
                    tail.compareAndSet(curTail, tailNext);
                } else {
                    if (curTail.next.compareAndSet(null, newNode)) {
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }
    public E take() {
        for (;;) {
            Node<E> oldHead = head.get();
            Node<E> oldTail = tail.get();
            Node<E> oldHeadNext = oldHead.next.get();
            if (oldHead == head.get()) {
                if (oldHead == oldTail) {
                    if (oldHeadNext == null) {
                        return null;
                    } else {
                        tail.compareAndSet(curTail, tailNext);
                    }
                } else {
                    if (head.compareAndSet(oldHead, oldHeadNext)) {
                        return oldHeadNext.item;
                    }
                }
            }
        }
    }
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Answer:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>T1</td>
</tr>
<tr>
<td>P2</td>
<td>T2</td>
</tr>
<tr>
<td>P3</td>
<td>T3</td>
</tr>
<tr>
<td>P4</td>
<td>T4</td>
</tr>
<tr>
<td>P5</td>
<td>T4</td>
</tr>
<tr>
<td>P6</td>
<td>T4</td>
</tr>
<tr>
<td>P7</td>
<td>T4</td>
</tr>
<tr>
<td>P10</td>
<td>T4</td>
</tr>
<tr>
<td>P10</td>
<td>T4</td>
</tr>
<tr>
<td>P10</td>
<td>T5</td>
</tr>
<tr>
<td>P10</td>
<td>T6</td>
</tr>
<tr>
<td>P10</td>
<td>T7</td>
</tr>
<tr>
<td>P11</td>
<td>T10</td>
</tr>
<tr>
<td>P12</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>