Sample Exam Problems For the Material in "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency"

Solutions Not Included

For more information, see http://www.cs.washington.edu/homes/djg/teachingMaterials.

Name:____

1. In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: A `String[]`
   - Output: A `Pair` of (a) the number of words starting with the letter `'c'` and (b) the length of the longest word starting with the letter `'c'`. (If no words start with `'c'`, the length is irrelevant.)

   Your solution should have $O(n)$ work and $O(\log n)$ span where n is the array length. Do *not* employ a sequential cut-off: the base case should process one `String`.

   We have provided some of the code for you. You only need to provide one class definition:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
class Pair {
  int count;
  int longest;
  Pair(int c, int l) { count = c; longest = l; }
}
class Main{
  static final ForkJoinPool fjPool = new ForkJoinPool();
  Pair processCWords(String[] array) {
     return fjPool.invoke(new ProcessCWords(array,0,array.length));
  }
}
```

Name:____

2. You are at a summer internship working on a program that currently takes 10 minutes to run on a 4-processor machine. Half the execution time (5 minutes) is spent running sequential code on one processor and the other half is spent running parallel code on all 4 processors. Assume the parallel code enjoys perfectly linear speedup for any number of processors.

   Note/hint/warning: This does *not* mean half the work is sequential. Half the *running time* is spent on sequential code.

   Your manager has a budget of $6,000 to speed up the program. She figures that is enough money to do only one of the following:

   - Buy a 16-processor machine.
   - Hire a graduate of this course to parallelize more of the program under the highly dubious assumptions that:
     - Doing so introduces no additional overhead
     - Afterwards, there will be 1 minute of sequential work to do and the rest will enjoy perfect linear speedup.

   Which approach will produce a faster program? Show your calculations, including the total *work* done by the program and the expected running time for both approaches.

Name:____

3. In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: An `int[]` (though the element type happens to be irrelevant)
   - Output: A new `int[]` where the elements are in the reverse order. For example, the element in the last array index of the input will be at index 0 in the output.

   Your solution should have $O(n)$ work and $O(\log n)$ span where n is the array length. Do *not* employ a sequential cut-off: the base case should process one array element.

   We have provided some of the code for you.

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Main {
    static final ForkJoinPool fjPool = new ForkJoinPool();
    static int[] reverse(int[] array) {
        // ADD A LINE HERE
        fjPool.invoke(new Reverse(answer,array,0,array.length)); // DO NOT CHANGE
        // ADD A LINE HERE
    }
}

// DEFINE A CLASS HERE
```

Name:____

4. Suppose a program uses your solution to the previous problem to reverse an array containing 2^{27} elements.

   (a) In English, about how big is 2^{27}? For example, "one thousand" is an answer in the right form, but is the wrong answer.

   (b) When the program executes, how many fork-join threads will your solution to the previous problem create? Give both an exact answer and an approximate English answer.

   (c) Suppose you modify your solution to use a sequential cut-off of 1000. When this modified program executes, how many fork-join threads will it create? Give both an exact answer and an approximate English answer.

Name:____

5. In class we learned an algorithm for parallel prefix sum with $O(n)$ work and $O(\log n)$ span. In this problem, you will develop similar algorithms for parallel *suffix* sum, where element `i` of the output array holds the sum of the elements in indices greater than or equal to `i` in the input.

   (a) Describe in English or high-level pseudocode a two-pass algorithm to solve the parallel suffix sum problem. Assume the reader is familiar with the parallel prefix sum problem, so for any parts that are *exactly* the same, you can just say they are the same. Any parts that are different need a full explanation.

   (b) Suppose you needed an implementation of parallel suffix sum, but you already had available a library for parallel prefix sum. Describe an easier-to-implement algorithm for parallel suffix sum that uses parallel prefix sum and one or more other algorithms as subroutines while maintaining $O(n)$ work and $O(\log n)$ span.

Name:____

6. (a) Suppose we have an algorithm for finding the second-smallest element in a binary search tree. The code is somewhat complicated to account for all possible tree shapes, but it always descends to the correct node without making any modifications to the tree. Suppose the code is in a method `secondSmallestMidterm`. Would it be correct for two threads to execute `secondSmallestMidterm` concurrently? Explain briefly.

   (b) Here is a simpler algorithm for finding the second smallest element in a binary search tree in terms of some other operations:

```
synchronized E deleteMin() { ... }
synchronized E findMin() { ... }
synchronized void insert(E x) { ... }
E secondSmallestFinal() {
  E min = this.deleteMin();
  E ans = this.findMin();
  this.insert(min);
  return ans;
}
```

   Notice `deleteMin`, `findMin`, and `insert` are synchronized methods.

   i. Suppose two threads call `secondSmallestFinal` concurrently. Demonstrate how one of them can get the wrong answer.

   ii. Does the code above have any *data races*? Explain briefly.

   iii. What is the easiest way to fix `secondSmallestFinal`?

Name:____

7. You are designing a new social-networking site to take over the world. To handle all the volume you expect, you want to support multiple threads with a fine-grained locking strategy in which each user's profile is protected with a different lock. At the core of your system is this simple class definition:

```
class UserProfile {
   static int id_counter;
   int id; // unique for each account
   int[] friends = new int[9999]; // horrible style
   int numFriends;
   Image[] embarrassingPhotos = new Image[9999];
   UserProfile() { // constructor for new profiles
     id = id_counter++;
     numFriends = 0;
   }
   synchronized void makeFriends(UserProfile newFriend) {
      synchronized(newFriend) {
        if(numFriends == friends.length
           || newFriend.numFriends == newFriend.friends.length)
          throw new TooManyFriendsException();
        friends[numFriends++] = newFriend.id;
        newFriend.friends[newFriend.numFriends++] = id;
      }
   }
   synchronized void removeFriend(UserProfile frenemy) {
      ...
   }
}
```

(a) The constructor has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

(b) The `makeFriends` method has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

(c) Rather than throwing an exception in `makeFriends` if an array is full, give two alternatives. Describe them only at a high level – a sentence or two is enough – without getting into any code details. One alternative should be easy and have nothing to do with concurrency. The other should involve concurrency.

Name:____

8. Answer “always” or “sometimes” or “never” for each of the following.

   (a) A program with data races also has bad interleavings.

   (b) A program with bad interleavings also has data races.

   (c) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has data races.

   (d) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has bad interleavings.

   (e) Java’s `synchronized` statement will block if the thread executing it already holds the lock that is being acquired.

   (f) This code snippet using a condition variable `foo` is a bug: `if(someCondition()) foo.wait()`.

Name:____

9. Consider an adjacency-list representation of an undirected graph where if edge (u, v) is in the graph then v is in u's adjacency list and u is in v's adjacency list. (This supports efficient operations for "find all nodes adjacent to a given node.") Suppose we wish to support multiple threads accessing the graph concurrently such that every thread always sees a consistent state of the graph. Further suppose we use a locking strategy where each adjacency list uses a different reentrant lock (one lock for the whole adjacency list).

   (a) This pseudocode for deleting an edge from the graph — and doing nothing if the edge is not already present — has a concurrency error. Write down an interleaving that exhibits a concurrency error and describe what happens when the interleaving occurs. Assume `array` holds the adjacency lists, vertices are represented by ints, and `contains` and `delete` are methods on lists (the latter throwing an exception if the item is not present).

   ```
   void deleteEdge(int u, int v) {
     synchronized(array[u]) {
       if(!array[u].contains(v))
         return;
     }
     synchronized(array[u]) { array[u].delete(v); }
     synchronized(array[v]) { array[v].delete(u); }
   }
   ```

   (b) This pseudocode also has a concurrency error. Repeat the previous problem with this code:

   ```
   void deleteEdge(int u, int v) {
     synchronized(array[u]) {
       synchronized(array[v]) {
          if(!array[u].contains(v))
            return;
          array[u].delete(v);
          array[v].delete(u);
       }
     }
   }
   ```

   (c) Write a correct version of `deleteEdge` in the same style as the broken versions above. For simplicity, you can assume no other operations modify the graph (or that they are written using the same approach as your method).