Name: _____

CMSC 433 Section 0101
Fall 2012
Midterm Exam #2

Directions: Test is closed book, closed notes, no electronics. Answer every question; write solutions in spaces provided. Use backs of pages for scratch work. By writing your name above, you pledge to abide by the University's Honor Code:

> **"I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination."**

Good luck!

Please do not write below this line.

---

1. _____	
2. _____	
3. _____	
4. _____	
5. _____	
SCORE _____	

1. (20 points) Answer each of questions in 1-2 sentences

   (a) (5 points) What is the Java Monitor Pattern, and what is it used for?

   The Java Monitor Pattern consists of making all fields private and all methods synchronized in a given class. This ensures that the resulting class is thread-safe.

   (b) (5 points) Why are Java programmers generally advised to use `notifyAll()` rather than `notify()` in their programs?

   Using notify() can lead to deadlocks if threads waiting on a given lock are waiting for different conditions; this is because notify() only awakens one thread. notifyAll() avoids this possibility, since every thread waiting is awakened and has a chance to proceed, once it reacquires the lock.

   (c) (5 points) Explain the difference between *balking* and *guarded suspension* approaches to implementing state-dependent actions.

   In a balking-based implementation, invoking a method (action) when the state does not support it results in a return value / exception being raised that signals the method could not complete normally. In guarded-suspension, when a thread invokes a method in a state that cannot allow normal execution, the thread instead goes to sleep and is reawakened when the state changes.

   (d) (5 points) What is "lock striping"?

   Lock striping is a method that uses multiple locks, rather than a single lock, to guard a data structure that is being used my multiple threads. As is the case with single-lock schemes, the goal is to ensure consistency of the data structure, even under multiple accesses; however, lock striping improves data-structure availability by using different locks to guard different parts of the data structure.

2. (20 points)

(a) (5 points) What is "nested monitor lockout"?

Nested monitor lockout can occur when one object has a synchronized method that calls a synchronized method of an inner, embedded object. In this case deadlock can occur if the thread calling the outer object's method enters the wait set of the inner object; the inner object's lock is released, but not the outer object's, meaning no other thread can access any synchronized method of the outer object.

(b) (7 points) Consider the following implementation of a thread-safe class of unbounded buffer of strings.

```
public class StringBuffer {
    private final ArrayList<String> strings;
    StringBuffer () { strings = new ArrayList<String>(); }

    public synchronized void put (String newString) {
        strings.add(newString);
        notifyAll();
    }

    public synchronized String take() throws InterruptedException {
        while (strings.size() == 0) { wait(); }
        String returnString = strings.get(0);
        strings.remove(0);
        return returnString;
    }
}
```

Now suppose we wish to implement a thread-safe string buffer that does not insert empty strings as follows.

```
public class StringBufferNonEmpty {
    private final StringBuffer buffer;
    StringBufferNonEmpty() { buffer = new StringBuffer(); }

    public synchronized void put (String newString) {
        if (!newString.equals("")) buffer.put(newString);
    }

    public synchronized String take () throws InterruptedException {
       return buffer.take();
    }
}
```

Give a situation in which nested monitor lockout can occur with this implementation of `StringBufferNonEmpty`.

Suppose a given StringBufferNonEmpty object obj is empty, and suppose a thread t1 executes obj.take(). obj.take() will in turn invoke the buffer.take() method of the obj's private buffer field. Since buffer is empty, t1 will enter buffer's wait set, releasing its lock on buffer. However, t1 still hold obj's lock, and no other thread can invoke any of obj's methods, including one that would insert a new element. t1 is therefore deadlocked.

(c) (8 points) How can the implementation of `StringBufferNonEmpty` be fixed so that nested monitor lockout cannot occur? (You may also alter the implementation of `StringBuffer` if you wish.)

One way to fix these classes is to parameterize the StringBuffer class on the lock that is used to synchronize its methods, and to modify StringBufferNonEmpty to supply the lock object used by the inner buffer class.

3. (20 points)

(a) (6 points) Suppose field `list` is initialized as follows.

```
List<Object> list = Collections.synchronizedList(new ArrayList<Object>());
```

Now consider the following method implemented in a separate class

```
public static void putIfAbsent (List<Object> l, Object o) {
    if (!l.contains(o)) l.add(o);
}
```

If `list` is initially empty, and two different threads invoke `putIfAbsent(list, o)` with the same object `o`, will `list` be guaranteed to contain only one copy of `o` after both terminate? Explain.

It is not guaranteed. The reason is that the two calls to the method putIfAbsent may interleave their respective checks of the presence of o in the list; if o is not their initially, both method calls may return true, and o will be added by each call.

(b) (7 points) Consider the variable `list` defined above, and assume that several threads are accessing it. Suppose one thread executes the following.

```
System.out.println(list);
```

Explain why the exception `ConcurrentModificationException` may be thrown by this statement.

This exception may be thrown because another thread could modify list while the thread is in the middle of printing the list. The println method uses list.toString(), which in turn uses iteration to construct strings for each element in the list. This iteration is not synchronized in the code above, which means that another thread may modify list while the iteration is in progress.

(c) (7 points) Give a fix to the statement in the previous part that eliminates the possibility of the given exception being thrown.

You need to synchronize on the list before the println().

4. (20 points)

(a) (5 points) Explain how bounded blocking queues may be used in Producer-Consumer applications to "slow down" producers to match the rate at at which consumers process elements from the queue.

When a bounded blocking queue is full, any calls a (Producer) thread makes to put() will cause the thread to be placed in the wait set for the queue, meaning the thread is suspended until a notify() / notifyAll() occurs. This thread can therefore not produce more elements while it is suspended. If the rate of production exceeds the rate of consumption, the net effect is therefore that more and more producers are suspended until the rates of production is approximately at the rate of consumption.

(b) (5 points) What mechanism do `CopyOnWriteArrayList` objects use to ensure that `ConcurrentModificationExceptions` can never be thrown?

CopyOnWriteArrayList objects create copies of the underlying list whenever a modification is made. The modification is made on the list copy, and then the new list replaces the old one. Any process that is iterating through a list will continue to iterate over the (old) list whenever one of these update iterations is applied, meaning that the iteration process never sees a change to the list, and thus the given exception is never thrown.

(c) (5 points) Explain the difference between a `CountdownLatch` object and a `Semaphore` object.

Both objects are synchronizers. In the case of a CountdownLatch, a thread executing an await() on such an object are blocked until the value of the Latch reaches 0, at which point all such threads are released an allowed to start executing. In the case of a semaphore object, threads are allowed to acquire permits and continue executing whenever the number of permits is greater than 0. When this value reaches 0, any thread trying to acquire a permit blocks until some thread releases its permit, at which point the thread (and others waiting) may try to acquire the newly released permit.

(d) (5 points) What is "thread-starvation deadlock?"

Thread-starvation deadlock occurs when a thread pool with a bounded number of threads are executing an equal number of tasks, and all tasks are blocked waiting for results of tasks that are in the work queue and not being executed. Because there are no threads available to execute these tasks, the system deadlocks.

5. (20 points) *Linear search* is an algorithm for locating the first position in an array at which a given element occurs. Linear search algorithms return this position, if the element occurs in the array, or -1 if the element is not in the array.

   Implement a parallel linear-search algorithm for locating an integer in an array of integers. Your method should have the following header.

   ```
   public static int find (int[] elts, int elt);
   ```

   Your solution must use an `Executor` of some sort, although the type you use is up to you. You should also make an attempt to "tune" your solution so that it efficiently exploits parallelism. For this purpose, you may find the method call `Runtime.getRuntime().availableProcessors()` useful.

   See project midterm2 in the CMSC 433 workspace.