

## 433 Final Exam Answers, May 20th

1. (25 points) Bakery algorithm for first-in, first-out queuing

This question requires you to reimplement the `OnePlaceBuffer` interface we used in our first day quiz. However, if multiple threads block trying to call `take()`, they should return in FIFO (first-in, first-out) order. Calls to `put(Object o)` do *not* have to complete in FIFO order. For our implementation, we are going to ignore and discard interrupts.

```
package cmsc433;
public interface OnePlaceBuffer {
    public Object take();
    public void put(Object o);
}
```

### ANSWER:

Here is my solution, based on the bakery algorithm:

```
public class MyBuffer implements cmsc433.OnePlaceBuffer {
    private Object buf = null;
    private int nowServing = 0;
    private int nextNumber = 1;
    public synchronized Object take() {
        int myNumber = nextNumber++;
        while(nowServing != myNumber)
            try {
                wait();
            } catch (InterruptedException e) {}
        Object result = buf;
        buf = null;
        notifyAll();
        return result;
    }

    public synchronized void put(Object o) {
        while(buf != null)
            try {
                wait();
            } catch (InterruptedException e) {}
        buf = o;
        nowServing++;
        notifyAll();
    }
}
```

A slightly different solution is:

```
public class MyBuffer implements cmsc433.OnePlaceBuffer
{
    private Object buf = null;
    private int nowServing = 0;
    private int nextNumber = 0;
    public synchronized Object take() {
        int myNumber = nextNumber++;
        while(buf == null || nowServing != myNumber)
            try {
                wait();
            } catch(InterruptedException e) {}
        Object result = buf;
        buf = null;
        nowServing++;
        notifyAll();
        return result;
    }

    public synchronized void put(Object o) {
        while(buf != null)
            try {
                wait();
            } catch(InterruptedException e) {}
        buf = o;
        notifyAll();
    }
}
```

Both of these solutions is fine. The only slight problem with it is that it uses notifyAll, and all of the waiting threads have to check to see if their number has been called.

Now, in class I said that this was usually not a problem in practice and no one would loose points in my class for doing a notifyAll solution. However, several students tried a solution using a queue of waiting threads, perhaps to avoid waking up all threads. This is very difficult to get right. There are all sorts of potential problems involving issues such as subtle race conditions, holding locks on more than one object, and deadlock.

However, for your enjoyment, here is my solution using a queue. In this solution, we only wake up one thread at a time.

```
public class MyBuffer2 implements cmsc433.OnePlaceBuffer {
    static class Status {
        boolean ready = false;
    }
    private Object buf = null;
    private LinkedList queue = new LinkedList();
    public Object take() {
        Status s;
        synchronized (this) {
            if (buf != null && queue.isEmpty()) {
                Object result = buf;
                buf = null;
                notify();
                return result;
            }
            s = new Status();
            queue.addLast(s);
        }
        synchronized(s) {
            while (!s.ready)
                try {
                    s.wait();
                } catch (InterruptedException e) {}
        }
        synchronized(this) {
            assert buf != null;
            Object result = buf;
            buf = null;
            notify();
            Status s2 = (Status) queue.removeFirst();
            assert s == s2;
            return result;
        }
    }

    public synchronized void put(Object o) {
        while(buf != null)
            try {
                wait();
            } catch (InterruptedException e) {}
        buf = o;
        if (!queue.isEmpty()) {
            Status first = (Status) queue.get(0);
            // holding two locks is OK because the locks are ordered
            // (we always get the Buffer lock before we get a Status
            // lock), and we don't do a wait while holding two locks
            synchronized(first) {
                first.ready = true;
                first.notify();
            }
        }
    }
}
```

2. (10 points) Iterating through collections in a multithreaded context.

Assume you have a collection `S` which is accessed and updated by multiple threads (protected by synchronization on `S`).

Write a function `foo()` that iterates through all of the elements of `S` and invokes `bar(Object o)` on each element of the collection.

Note that a call to `bar` could take a while to perform.

Discuss any particular advantages/disadvantages/features of your solution that a user of your code should be aware of.

**ANSWER:**

Since a call to `bar` could take a while to perform, we don't want to hold a lock on `S` for the duration of all the calls to `bar`. However, we are not going to perform the calls to `bar` in parallel. This solution makes a copy of the collection before the iteration starts, and thus any changes to the collection during the iteration will not be reflected in the iteration.

```
void foo() {
    LinkedList lst;
    synchronized(S) {
        lst = new LinkedList(S);
    }
    for(Iterator i = lst.iterator(); i.hasNext(); )
        bar(i.next());
}
```

3. (10 points) XML. There are two basic approaches/APIs for dealing with XML documents. In class, we discussed these as the SAX and the DOM approaches. Briefly describe the differences and relative advantages/disadvantages of these approaches.

**ANSWER:** The SAX approach essentially provides a visitor pattern over the XML tree. Callbacks are made at each internal and leaf node.

The DOM approach generates a tree data structure representing the entire XML tree.

The SAX approach can be more efficient, particular in terms of memory. However, the DOM approach allows arbitrary traversals and modifications of the tree.

4. (5 points) Security. Why might you wish to restrict the machines and ports that untrusted code can connect to?

**ANSWER:** Because untrusted code might be run behind a firewall. Machines behind a firewall are often not as secured as machines directly exposed to the internet, and often provide unsecured services. For example, untrusted code could problem local machines to see if any were vulnerable to well-known but unfortunately common vulnerabilities (such as Windows XP having a default administrator account with no password).

5. (5 points) You've calculated/measured that the data structures in use by your program require  $100+10n$  Kbytes at any one time to handle  $n$  warehouses. Your program will be doing a reasonable amount of memory allocation (with some of the previously allocated memory becoming unreachable and therefore garbage), so you need to worry about the cost of garbage collection.

Assume you are using a standard stop-and-copy garbage collector. For 1000 warehouses, which number is best estimate for the amount of heap memory your program would need to run in so that garbage collection would be efficient? Explain your choice:

- 10 Megabytes

- 15 Megabytes
- 25 Megabytes
- 50 Megabytes

**ANSWER:** 25 Megabytes

Your data structures need 10.1 Megabytes of live memory. With a simple copying collector, you have two semispaces, each large enough to hold all of live memory and a little extra. So 25 Megabytes is enough, and 50 Megabytes is far more than you need.

6. (10 points) Mix and match

For each item in the left column, match it with the problem or domain in the right column for which it would most likely be useful.

Each item should be paired no more than once. *There is one item in each column that should not be paired at all (in other words, there is one bogus entry in each column).*

**ANSWER:**

Technology		Problem
Interrupts	1	g Shutting down
Multicast	2	d Finding peers
Observer pattern	3	e GUIs
Reflection	4	b Debuggers and IDEs
Visitor pattern	6	a Abstract syntax trees
XML	7	f Saving state

7. (15 points) Factory methods There are two basic approaches to allowing users to create instances:

- public constructors
- public static factory methods and non-public constructors

Give the relative advantages/disadvantages of these two approaches.

**ANSWER:** Public constructors have standard naming conventions, and make it easy to extend the class.

Static factory methods don't have to create a new object each time (e.g., *Boolean.valueOf(boolean b)*), and can return instances of different types (e.g., the *InetAddress* factory methods can return either an *Inet4Address* or a *Inet6Address*).

On the exams, a lot of students confused static factory methods with the singleton pattern. Which static factory methods can be used to implement the singleton pattern, they are many other uses for static factories.

8. (20 points) Distributed computing:

One of the design choices in RMI was to force every method in a remote interface to throw a *RemoteException* and make *RemoteException* a declared exception (so that if you call a method that could throw a remote exception, you must either catch the exception or declare that you could throw it).

An alternative would be to make it be a *RuntimeException*, like a *NullPointerException*.

Discuss whether the decision made in RMI is a good or bad decision, and why. Feel free to refer to the ideas discussed in the *A Note on Distributed Computing*.

**ANSWER:** It is a good decision, because remote communication is unreliable by its very nature. It is reasonable to write code that expects that null pointer exceptions won't occur, but it isn't reasonable to assume communications with remote machines is reliable. Since all calls to a remote reference are declared as throwing a remote exception, any code making such calls must anticipate the potential failure and decide what to do in that case (of course, you can always just catch and ignore the exception, but that is a decision too).