

## 433 Practice 2nd Midterm

### 1. (Threading and locks)

- (a) Define a Java class Room. When a room is created, you specify the capacity (an integer). Implement two functions, enter() and leave(). Keep track of the number of threads that have entered a room. If the room is at capacity, a new thread trying to enter the room will block until somebody leaves and the thread can enter without pushing the room over capacity.  
Don't worry about handling recursive locks.
- (b) Provide an additional function doInRoom(Runnable r) that enters the room, invokes the run method of r, and then leaves the room. The leave() method is invoked even if the run method of r throws an exception (although execution of doInRoom still terminates with the exception thrown by the run method).

#### Answer:

```
public class Room {
    int capacity;
    public Room (int c) {
        capacity = c;
    }
    public synchronized void enter() throws InterruptedException {
        while (capacity == 0) wait();
        capacity--;
    }
    public synchronized void leave() {
        if (capacity == 0) notify();
        capacity++;
    }
    public void doInRoom(Runnable r) {
        // Ignore InterruptedException
        while (true) {
            try { enter(); break; }
            catch (InterruptedException e) {}
        }
        try {
            r.run();
        }
        finally {
            leave();
        }
    }
}
```

2. Write a class FIFOmonitor (FIFO = first in, first out). It should have two functions, acquire() and release(). Threads should acquire locks in the order in which they invoke the acquire function. The acquire function should ignore interrupts.

#### Answer:

```
public class FIFOmonitor {

    private int nowServing = 0;
    private int nextNumber = 0;
    public synchronized void acquire() {
        int num = nextNumber++;
    }
}
```

```

        while (num < nowServing) {
            try { wait(); }
            catch (InterruptedException e) {}
        }

    public void release() {
        nowServing++;
        notifyAll();
    }
}

```

3. Why does Sun now discourage the use of `Thread.stop()`? Briefly describe alternatives.

**Answer:** `Thread.stop()` abruptly stops a thread. If that thread has some data structure locked and is part way through making a change, the data structure may be in an inconsistent state at the time the thread is stopped.

Although it is theoretically possible to try to program defensively, it is very difficult because you have no idea of the place where your thread might die.

Alternatively, you can use `interrupts` or set a volatile boolean flag indicating that you wish the thread to stop. If you use `interrupts`, then the thread has to respond properly to `interrupts` and to occasionally check for `interrupts`. If you use a volatile flag, then you need to occasionally poll the flag.

4. Say a server wanted to be able to send a stream of messages to a client. You want to make sure that the client receives the messages in order, without skipping any. Allow for the machine the client is running on to crash, with the client migrating to another computer.

Describe the high level design of a system to handle these goals.

**Answer:** You need to assign sequence numbers to each message so that missing, duplicate and/or out-of-order messages can be detected.

There needs to be some sort of log-on authentication for a client so that if a client reconnects or moves, the server can be sure that it is authorized.

The server should hold onto old messages, so that if a client moves, the server can resend any messages that were not committed by the client before the crash/move.

The server API needs to provide methods both for reconnection and for requesting old messages.

5. Why does a remote object need to implement a remote interface? Alternatively, what is special about a remote interface?

**Answer:** The remote interfaces tell the RMI what methods to build stubs for and which interfaces the stub should implement.

Also, if code wishes to reference a remote object, it cannot do so with a variable whose type is that of the remote object (because then the variable cannot reference the stub). Instead, variables used to reference remote objects should have a type that is one of the remote interfaces implemented by the remote object.

6. Jini has something called *leasing*. Describe it and why it is useful.

**Answer:** For the environment Jini is design to work in, it is unrealistic to expect devices to logoff when they are turned out, crash, or leave the network. Instead, devices must “phone home” on a periodic basis to let people know they are still available. The appropriate time period for leasing is a trade off between communication overhead and how accurate the information is.

7. Say you had implemented a `HashMap` class (ignore the one already provided in the JDK). Is it reasonable for you to declare that your class is `Serializable`? What would the default serialization do to a `HashMap`? What if some values appeared multiple times?

If you wrote a custom serialization, how might you do it differently to make serialization more efficient?

It is reasonable to declare it as `Serializable`. The standard semantics would be to just capture the structure of the entire `HashMap`, along with all of the entries. Any duplicate values would correctly be serialized as a single value, referenced multiple times.

For a custom serialization, it would be more efficient to simply encode the hash table size, the number of entries, and then the key-value pairs.

8. Assume you would like to instantiate a `Set` implementation twice; once to provide a set of `ints` and again to provide a set of `Strings` using C++ templates or Generic Java (GJ). Would this work in both languages? If not, why not? What are the language design choices that led to this, and what are some of the other impacts of those design choices? fields Such an implementation would work with C++, but not with GJ, because in GJ it's not possible to give a non-Object type (`int`) as a type parameter.

C++ can allow primitive types as type parameters because the compiler instantiates different machine-specific object code for each template instantiation, so can deal with basic types as well as user-defined class types.

Because GJ translates into Java byte codes that only deal with `Object` (or some subclass of `Object`) references, GJ can produce one set of byte codes per generic class, instead of C++ producing separate code for each instantiation of a templated class. The GJ design also allows for easy integration of generic with non-generic classes and code (new-style and old-style), whereas C++ template code will not work with non-template versions of the classes (and vice versa).

Another reason C++ templates can work with primitive types or class types is that C++ allows operator overloading, effectively allowing primitive and class types to work in many cases with the same syntax (if the class provides the desired overloaded operators). Only a few Java operators work with both primitive and class types (e.g., `=`, `==`, `!=`), and Java does not support operator overloading.