CSCI 355 LAB #8 Spring 2007

Concurrency in Distributed Objects

Objectives:

1. Explore concurrency in distributed objects.
2. Use of `synchronized` keyword in Java.

Preparation: Before lab read pages 1052-1076; 1093-1100 in *Java: How to Program* by Deitel and Deitel, sixth edition, 2005

Laboratory Assignment:

The expected solution to the distributed object `Bank` program of Lab 7 has a subtle design flaw.

Before we explore this design flaw, let us introduce some background. You discovered in the last lab that the implementation of the Bank object interface allows several customers to remotely access the object at the same time, i.e., **concurrently**. The Java RMI system creates a new **thread** for each client's remote invocation. After the return of the invocation, the thread is released. If two clients call a remote method at the same time, the two threads for the clients share the same code of the method and instance variables.

Whenever, two or more threads (or processes) share the same variables, we have the potential for interference and, hence, corruption of the computation. This is the source of the above mentioned design flaw!

1. Analyze the Code for Shared variables

Copy your distributed object `Bank` solution from Lab 7 to your Lab 8 directory. Analyze the code for shared variables.

Each thread in Java has it's own private activation stack. Therefore, no local objects are shared. All the threads share any static (class) variables and object instance variables.

List the shared variables found in your Lab 7 solution:

____

____

2. What are the Critical Regions?

To protect the shared variables from being corrupted, we allow at most one thread at a time to execute specific segments of code involving the shared variables called **critical regions** (or critical sections). We say that we want the critical regions to be executed **mutually exclusively** (mutual exclusion property).

We must worry about two threads updating a shared variable at the same time. Also, we must worry about one thread using a shared variable while another thread updates it. We have to analyze each potential situation.

Analyze the code for critical regions. On a printout of the implementation of the Bank object interface (probably called `BankInterfaceImpl.java`) file you should bracket with a pencil the critical regions.

3. The `synchronized` Keyword

Java provides the keyword `synchronized` to protect critical regions which can be a whole method or a block of code.

Unfortunately, in Java, a *remote* method may *not* be `synchronized`! You will need to use the second approach.

Modify the Bank program to use the `synchronized` keyword to protect the critical regions.

Be careful using the `synchronized` keyword. There is an overhead associated with `synchronized` and using it forces concurrent process to execute `synchronized` code one at a time. Therefore, using `synchronized` where it is not needed can slow performance significantly.

4. Constructors Can Never Be Called Remotely

A constructor of a remote object can **NEVER** be called remotely, i.e., it can not be in the interface. This is true in Java RMI and in other distributed object systems such as CORBA. Try to think about why this is true.

In light of the above exercises, explain why constructors are never allowed to be called remotely. Hint: What could happen if this were allowed?

____

____

In fact in Java, a constructor can never be in any interface not just remote ones.

5. Reflecting on the Behavior of Bank Server

Look at the code for your Bank server. This is the class which calls the `reg.rebind()` method to create the Bank object. The code is short and very straight forward.

However, when you run this class observe what happens. The program never exits! Why?

____

____

Hand In:

Hand in a copy of the listing of all the modified Java files from Exercise 3 and the answers to the questions of Exercises 1, 2, 4 and 5.