Threads 4 / RMI

Semaphore1

Semaphore1 from last time uses the count in a precise way to know exactly how many threads are waiting. In this way, the notify() is never dropped.

Conservation of notify

The notify() "conserves" the permission to go -- passing from one thread to another.

if wait

We use if() logic on the waiting side -- when the notify comes through, one waiter can unblock, even though the count may still be negative.

Semaphore2

More straightforward implementation -- use the classic while-wait structure. decr() does not make the count negative. notify() is not conserved, but the logic is much simpler.

Semaphore2 Code

```
/*
 Alternate, more readable implementation of counting Semaphore --
 uses the classic wait pattern:
   while (!cond) wait();

 In this version, decr() does not move the count < 0,
 although the semaphore may be constructed with a negative
 count.
 This version is slightly less precise than above,
 since the notify() does not know if there is a matching
 wait(). This is not a big deal -- a notify() with no matching
 wait() is cheap. The precise semaphore counts the waits(), so
 it's notify() has a matching wait().
*/
class Semaphore2 {
   private int count;

   public Semaphore2(int value) {
      count = value;
   }

   // Try to decrement. Block if count <=0.
   // Returns on success or interruption.
   // The Semaphore value may be disturbed by interruption.
   public synchronized void decr() {
      while  (count<=0) try {
         wait();
```

```
      }
      catch (InterruptedException inter) {
         Thread.currentThread().interrupt();
         return;
      }
      count--;
   }

   // Increase, unblocking anyone waiting.
   public synchronized void incr() {
      count++;
      notify();
   }
}

/*
 Use two Semaphores to get A and B to take turns.
 This code works.
*/
class TurnDemo2 {
   Semaphore2 aGo = new Semaphore2(1);    // a gets to go first
   Semaphore2 bGo = new Semaphore2(0);

   // <cut>

   public void demo() {
      final Semaphore2 finished = new Semaphore2(-1);
      new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) {b(); }
            finished.incr();
         }
      }.start();

      new Thread() {
         public void run() {
            for (int i = 0; i< 10; i++) {a(); }
            finished.incr();
         }
      }.start();


      finished.decr();
      System.out.println("All done!");
      // Alternative: init Semaphore to 0 and decr() twice
   }
}
```

Q1: if vs. while

Q: What if the decr() used an if instead of a while?

A: This would suffer from barging: another thread makes the count 0 in-between when the notify() happens and the wait() unblocks.

Q2: if (count==1) notify();

Q: what if the incr() tried to be clever and only do the notify when making the 0->1 transition.

A: this won't work because we might have three threads blocked at count==0. Suppose incr() happens three times. We need three notifies, not just one.

Q3 Interruption?

Q: When decr() returns, do you have the lock or not?

A: These implementations, you may or may not have the lock. Suppose the interrupt comes through not in the wait() but in the count--. It would be possible to write the Semaphore so it returned something from decr() to indicate if the lock is now held, but this makes the client side more messy everywhere.

Wait for threads to finish

The demo example now has a "finished" Semaphore that the launcher uses to wait for the children. Alternately, could init the Semaphore to 0 and decr twice. Or, could use join() twice.

Remote Method Invocation

Interact with objects on other machines as if they were local
Local "stub" object -- proxy for real remote object

Advantages

Sockets

Easier than sockets -- just looks like message send

Simple

Scales -- you can interact with an object on your machine or somewhere else with practically the same code.

Performance: OK, not great

Doing your own socket based communication would be faster.

CORBA

CORBA is a language-neutral, platform-neutral -- things are expressed in the language independent Interface Description Language (IDL)
CORBA provides lots of data transport, but does not move code
CORBA is partly useful, and partly it's a management check-off item since it gives the appearance of portability and replaceability
RMI provides consistency by just using Java everywhere
RMI can actually move code back and forth -- Corba handles cross-language compatibility, but it does not move code from one place to another.

JINI

JINI is very much based on the idea of "mobile code". Your CD player sends UI code that presents the CD players interface to your Palm Pilot. The UI code then runs on the Palm. In this way, the Palm works with all devices -- even ones it does not know about.

==RMI Structure

FooRemote Interface

Interface off java.rmi.Remote
Everything throws RemoteException
Defines methods visible on client side
The client will actually hold a "stub" object that implements FooRemote
Client messages on the stub get sent back to the real server object.

FooServer

Subclass off UnicastRemoteObject
Implement FooRemote

This is the "real" object
Implements the messages mentioned in FooRemote
Can store state and implement other messages not visible in FooRemote

Live / Remote vs. Serialization

Remote

Remote objects use RMI so there really is just one object.
Messages sent on the remote stub tunnel back to execute against the one real server object.

Non-Remote = Serialized

Non remote arguments and return values use serialization to move copies back and forth.

rmic tool

Looks at the .class for the real object (FooServer) and generates the "stub" and "skel" classes used by the RMI system
rmic FooServer -> produces FooServer_Stub.class and FooServer_Skel.class
User code never mentions these classes by name-- they just have to be present in runtime space of the client and server so the RMI impl can use them.

Stub

Used on the client side as a proxy for the real object

Skeleton

Used on the server side to get glue things back to the real object

Which Classes in Which Runtime

Client:

FooRemote, FooServer_Stub

Server:

FooRemote, FooServer, FooServer_Stub, FooServer_Skel (i.e. everything)

One directory

Our low-budget solution: build and run everything in one directory, but launch the client and server jobs on separate machines.
Do not need a CLASSPATH set at all -- we'll rely on the "current directory" for both server and client

Abbreviated Code

FooRemote

```
public interface FooRemote extends java.rmi.Remote {

   public String[] doit() throws RemoteException;
   public void install(PipeRemote pipe) throws RemoteException;
```

Foo Server

```
public class FooServer extends UnicastRemoteObject
      implements FooRemote
{

   ...

    public String[] doit() throws RemoteException {
      Log.print("FooServer: doit start");
      serverInternal();

      if (pipe != null) pipe.send("Sent on the pipe");

      String[] result = new String[2];
      result[0] = "hello";
      result[1] = " there";
      return(result);
   }

   ...
}
```

/ / Client.java

```
import java.rmi.*;
import java.math.*;

public class Client {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         String name = "//" + args[0] + "/" + FooRemote.SERVICE;

         FooRemote foo = (FooRemote) Naming.lookup(name);

         String[] result  = (String[]) foo.doit(); // key line


         Log.print("Client: result ID " + result.hashCode());
         Log.print("Client: received from server -- " + result[0] + result[1]);

         PipeRemote pipe = new PipeServer();
         Log.print("Client: pipe ID " + pipe.hashCode());

         foo.install(pipe);

         foo.doit();
      } catch (Exception e) {
         System.err.println("FooClient exception: " +
            e.getMessage());
         e.printStackTrace();
      }

      Log.print("Client: done");
   }
}
```

PipeServer / PipeRemote

Server on client

Send to server

Server messages its PipeRemote, it executes back here on the client where the real PipeServer is held.