CS193k, Stanford
Spring, 2000-01

Handout #10
Nick Parlante

Threads 4 / RMI

Semaphore2

Alternate implementation -- possibly more readable. Does the wait/decrement in a different order. Uses the classic while-wait structure. The count does not go negative here -- so it does not count how many waiting threads there are. As a result, the notify() may happen at times when it is not necessary -- no big deal.

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 our first version,
 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();
   }
```

}

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.

interrupt() vs. wait(), sleep(), ...

- When blocked in wait(), sleep() or possibly I/O, an interrupt throws out of the blocking.
- The caller needs to realize if they got control normally, or because of an interrupt.
- Use a try/catch
- Unfortunately, in the catch, isInterrupted is **no longer true**. A simple strategy is to re-assert interrupt() so that other parts of the code can see that interruption has happened.
- From the Semaphore1 code...

```
public synchronized void decr() {
   count--;
   if (count<0) try{
      wait();
   }
   catch (InterruptedException inter) {
      // This exception clears the "isInterrupted" boolean.
      // We reset the boolean to true so our caller
      // will see that interruption has happened.
      Thread.currentThread().interrupt();
   }
}
```

Swing Threading

1. One Swing Thread

There's a special Swing thread (we'll think of it as one thread, although it could be several threads cooperating with locks)

Dequeues real time user events

Translates to paint() and action notifications

Once a swing component is subject to pack() or setVisible, no other thread should send it Swing sensitive messages such as add(), setPreferredSize(), getText ...

Except these four messages may be sent from any thread: repaint(), revalidate(), addXXXListener(), removeXXListener()

2. Swing Notifications -- OK

Any notification, such as actionPerformed() is sent on the Swing thread. Therefore, your notification code is running on the Swing thread, so you are allowed to message any swing state from there.

3. One (fast) Thing at a time

The swing thread does one thing to completion, and then does the next thing.

Therefore, you **never** get concurrent access problems between two operations being run on the swing thread. Synchronization is not required.

Just don't do something that blocks or takes a long time -- for something costly, create a separate thread and have report back (see below) when it has something.

4. Enqueue For The Swing Thread

If you are not in the Swing thread, get the Swing thread to do something for you. The following utilities enqueue something for the Swing thread to do for you when it next dequeues a job...

Returns immediately: SwingUtilites.invokeLater(new Runnable() { public void run() { ...}

Blocks: SwingUtilities.invokeAndWait(new Runnable() { ...

SwingThread.java

```
// SwingThread.java
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
import java.awt.event.*;

/*
 Demonstrates the affects of the one swing thread.
 1. Notifications, such as actionPerformed() are done on the
 Swing thread.
 2. Never get concurrent access problems with two
 swing thread actions. They inherently go one at a time.
 3. Swing timer class to send periodic messages on the
 Swing thread.
 4. Doing a long computation on the Swing thread locks
 up the GUI.
*/
public class SwingThread {

   public static void main(String[] args) {
      JFrame frame = new JFrame("SwingThread");
      JComponent container = (JComponent) frame.getContentPane();
      container.setLayout(new BorderLayout());

      /*****
       A
       Simple case -- connect a widget to a button.
      */
      final Widget a = new Widget(100, 100);
      JButton button = new JButton("Increment");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) { // on swing thread here
            a.increment();
         }
      });

      JPanel panel = new JPanel();
      panel.add(button);
      panel.add(a);
      container.add(panel, BorderLayout.NORTH);


      /*
       B
       Increment button w/ timer thread also sending increment --
       no conflict since both on Swing thread.
      */
      final Widget b = new Widget(100, 100);
      button = new JButton("Increment");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            b.repaint();   // 2 unnecessary calls to repaint
            b.increment();
            b.repaint();
         }
      });

      Box box = new Box(BoxLayout.Y_AXIS);
      box.add(button);
      box.add(b);

      // create timer for b
```

```
      final javax.swing.Timer timer =
         new javax.swing.Timer(500, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               b.increment();
            }
         });

      // start/stop button for the timer
      button = new JButton("Start/Stop");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            if (timer.isRunning()) timer.stop();
            else timer.start();
         }
      });
      box.add(button);

      container.add(box, BorderLayout.CENTER);

      /*
       C
       Demonstrates occupying the swing thread
       so nothing else works.
      */
      final Widget c = new Widget(100, 100);
      button = new JButton("C");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            c.increment();
            for (int i=0; i<150000000; i++) {}  // occupy the swing thread
            c.increment();
         }
      });

      panel = new JPanel();
      panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
      panel.add(button);
      panel.add(c);
      container.add(panel, BorderLayout.SOUTH);

      frame.pack();
      frame.setVisible(true);

   }

}
```

ThreadGui

Demonstrates launching worker threads to do actual work. They communicate back to the main thread through the SwingUtilities. The lanucher thread can detect when all the workers are finished, so it can set the state of the buttons back.

```
// ThreadGui.java
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

import java.awt.event.*;
import java.util.*;

import java.net.*;
import java.io.*;

import javax.swing.table.*;
import javax.swing.event.*;
import javax.swing.text.*;
/*
 A simple program which demonstrates using threads
 to do a long computation separate from the Swing thread.
 Uses a progress bar and stop button.
*/

/*
 Worker Thread class -- does actual work.

 -Messages the model object to do work
 (the model messages are thread safe).

 -Messages the GUI to update it with the
 current status.
*/
class Worker extends Thread {
   WFrame frame;
   IntModel model;
   boolean success;

   public static final int MAX = 1000000;
   public static final int PROGRESS_COUNT = 100;   // number of updates we get from
each thread


   public Worker(ThreadGroup group, WFrame frame, IntModel model) {
      super(group, "Worker");

      this.frame = frame;
      this.model = model;
   }


   public void run() {
      frame.threadChange(1);  // inform we've started
```

```
      setPriority(getPriority() -1);   // Not necessary -- may improve GUI
responsiveness

      int progressModulo = (int) MAX/PROGRESS_COUNT + 1;


      // Loop around and...
      // -do work on the model
      // -check for interruption
      // -message the GUI about progress
      for (int i=0; i<MAX; i++) {
         model.changeValue(1);   // do actual work

         // cheap interrupted test
         if ((i%40)==0 && isInterrupted()) break;

         // more expensive Progress notification
         if ((i%progressModulo) == 0) {
            SwingUtilities.invokeLater(   // safe way to call any GUI method
               new Runnable() {
                  public void run() {
                     frame.workerProgress();
                  }
               }
            );
            yield(); // polite to do this moderately often
         }

      }

      frame.threadChange(-1); // inform we're done
   }
}

/*
 Simple, thread safe data model that the GUI owns
 and the threads pound on.
*/
class IntModel {
   int value;

   public IntModel(int value) {
      this.value = value;
   }

   public synchronized int getValue() {   // NOTE get followed by set is not thread
safe
      return(value);
   }

   public synchronized void setValue(int value) {
      this.value = value;
   }

   public synchronized void changeValue(int delta) {  // NOTE but this is thread
safe
      value += delta;
   }
```

```
}


class WFrame extends JFrame  {
   public static final boolean JOIN = true;  // Controls which end strategy is used
                                    // I now prefer the "join" strategy

   IntModel model;
   JComponent container;

   JButton startButton, stopButton;
   JProgressBar progress;
   ThreadGroup threadGroup;

   // The current state
   int toBeDoneThreads;
   int doneThreads;
   int runningThreads;
   int aliveCount;
   JLabel label;
   Vector threads;


   public WFrame(String title) {
      super(title);


      model = new IntModel(0);
      runningThreads = 0;
      aliveCount = 0;

      container = (JComponent)getContentPane();
      container.setLayout(new BorderLayout(6,6));

      Box box = new Box(BoxLayout.Y_AXIS);
      container.add(box, BorderLayout.CENTER);


      JButton button = new JButton("Start");
      box.add(button);
      button.addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               userStart();
            }
         }
      );

      startButton = button;

      label = new JLabel(" ");
      box.add(label);

      progress = new JProgressBar();
      box.add(progress);

      button = new JButton("Stop");
```

```
        box.add(button);
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    userStop();
                }
            }
        );
        stopButton = button;


        pack();
        setVisible(true);
    }





    /*
     The launcher.
     Init the thread tracking state, and then launch a thread to launch
     all the workers. All the threads are in a thread group to make interruption
     convenient.
    */
    public void userStart() {
        final int WORKERS = 8;

        threadGroup = new ThreadGroup("Worker threads");

        // Here we are the Swing thread, so we can touch all
        // this state safely.
        stopButton.setEnabled(true);
        startButton.setEnabled(false);


        // This state is mostly for the non-JOIN way
        toBeDoneThreads = WORKERS+1;  // how many threads need to finish for us to be
"done"
        doneThreads = 0;
        aliveCount = 0;
        runningThreads = 0;

        progress.setMaximum(WORKERS * Worker.PROGRESS_COUNT);
        progress.setValue(0);

        threads = new Vector();

        new Thread(threadGroup, "Worker Launcher") {
            public void run() {
                threadChange(1);  // inform we're starting

                // start all the worker threads
                int i;
                for (i = 0; i< WORKERS; i++) {
                    if (isInterrupted()) {
                        // TRICKY update the goal if we are exiting early
                        toBeDoneThreads = i+1;
```

```
                break;
             }
             // "join" strategy
             if (WFrame.JOIN) {
                Worker worker = new  Worker(threadGroup, WFrame.this, model);
                threads.addElement(worker);
                worker.start();
             } else {
             // "int" strategy
                Worker worker = new Worker(threadGroup, WFrame.this, model);
                worker.start();
             }
          }


          // Here is the JOIN strategy to wait for the workers to finish
          if (WFrame.JOIN) {
             for (i =0; i< threads.size(); i++) {
                try{
                   ((Worker)threads.elementAt(i)).join();
                }
                catch (InterruptedException ignored) {}
             }

             // Set the GUI back when all the workers have exited
             SwingUtilities.invokeLater(
                new Runnable() {
                   public void run() {
                      threadDone();
                   }
                }
             );
          }

          threadChange(-1); // inform we're finishing
       }
    }.start();

 }


 /*
  The stop button has been clicked -- go interrupt all the threads.
  In 1.2 you can write threadGroup.interrupt().
  In 1.1 it needs to be written as below.
 */
 public void userStop() {
    Thread threads[] = new Thread[100];
    int len = threadGroup.enumerate(threads);
    for (int i=0; i<len; i++) {
       threads[i].interrupt();
    }
 }


 /*
  Sent by the threads when they start and top.
  +1 == starting
```

```
     -1 == ending
     This is a "safe" method.
     Maintains the runningThreads label in the GUI
     Notices when all the threads have finished and
     update the GUI (join() in the launcher is a better
     way to detect that all the threads are done).
    */
    public synchronized void threadChange(int delta) {
       runningThreads += delta;
       SwingUtilities.invokeLater(
          new Runnable() {
             public void run() {
                label.setText("" + runningThreads);
             }
          }
       );

       if (delta == -1) {
          doneThreads++;

          if (!WFrame.JOIN && doneThreads == toBeDoneThreads) {
             SwingUtilities.invokeLater(
                new Runnable() {
                   public void run() {
                      threadDone();
                   }
                }
             );
          }
       }
    }

    // Reset the GUI to the "threads are done" state
    public void threadDone() {
       progress.setMaximum(0);
       progress.setValue(0);
       stopButton.setEnabled(false);
       startButton.setEnabled(true);
    }


    // Sent by the threads periodically
    // This is not a safe method, the worker is responsible
    // for safety (see Worker).
    public synchronized void workerProgress() {
       aliveCount++;
       progress.setValue(aliveCount);
    }


 }

 public class ThreadGui {

    public static void main(String args[]) {
       new WFrame("Thread Gui");
    }
 }
```

Remote Method Invocation - RMI

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;
```

Foo Server

```
public class FooServer extends UnicastRemoteObject
      implements FooRemote
{

   ...

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

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

   ...
}
```

// Client.java

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

public class Client {
   public static void main(String args[]) {

      try {

         // snip setup

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

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


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

   }
}
```