Threading 3

Thread Challenge #2 -- wait/notify Co-ordination

Synchronization is the first order problem with concurrency. The second problem is co-ordination -- getting multiple threads to co-ordinate their schedules.

Checking condition under lock

Suppose you want to execute the statement "if (len >0) len++;" but other threads also operate on len.

Acquire the lock first, then look at len -- otherwise some other thread may change len in between the test and the len++

Do operations with the lock so the data is not changing out from under you -- this is just a basic truism of threads that read and write shared data.

wait() and notify()

Every Java Object has a wait/notify queue that threads can use to block efficiently waiting for a signal.

You must have that object's lock first before doing any operation on its queue (the queue is like "len" in the above example)

Use the wait/notify queue coordinate the actions of threads -- get them to cooperate and signal each other

wait()

obj.wait();

Send to any object -- the calling thread waits (blocks) on the object's queue

"Suspend" on that object's queue -- efficient blocking

Must first have that object's lock (or get a runtime error)

The waiting thread releases that object's lock (but not other held locks)

an interrupt() sent to the waiting thread will pop it out of its wait()

notify() / notifyAll()

obj.notify(); obj.notifyAll()

Send to any object -- notifies a waiter on that object's queue, if there is one

The sender must first have the object lock

A random waiting thread will get woken out of its wait() when the sender releases the lock. Not necessarily FIFO. Not right away.

The waiter will re-acquire the lock before resuming (this is part of what makes the random lag between the notify and the resume)

"dropped" notify

if there are no waiting threads, the notify() does nothing

wait()/notify() **do not count up and down** to balance things -- you need to build a Semaphore for that feature

variant: notifyAll() notifies all the waiting threads, not just a single one

In the simplest case, just use notifyAll(). It's tricky to know when notify() is sufficient. NotifyAll() is simpler, and it's not really slower in most cases.

Monitor Exception

java.lang.IllegalMonitorStateException: current thread not owner

This is the exception that is thrown if a thread tries a wait/notify on an object without first holding its lock

barging / Check again

When coming out of a wait(), check for the desired condition again -- it may have become false again in between when the notify happened and when the wait/return happened.

while

Essentially, the wait is always done with a while loop, not an if statement.

wait/notify Example

```
/*
 Producer/Consumer problem with wait/notify
 This code works correctly.

 -"len" represents the number of elements in some imaginary array
 -add() adds an element to the end of the array. Add() never blocks --
 we assume there's enough space in the array.
 -remove() removes an element, but can only finish if there
 is an element to be removed. If there is no element, remove()
 waits for one to be available.

 Strategy:
 -The AddRemove object is the common object between the threads --
 they use its lock and its wait/notify queue.
 -add() does a notify() when it adds an element
 -remove() does a wait() if there are no elements. Eventually,
 an add() thread will put an element in and do a notify()
 -Each adder adds 10 times, and each remover removes 10 times,
 so it balances in the end.
*/
class AddRemove {
   int len = 0;   // the number of elements in the array
   final int COUNT = 10;

   public synchronized void add() {
      len++;
      System.out.println(Thread.currentThread().getName() + " add " + (len-1));
      notifyAll();
   }

   public synchronized void remove() {
```

```
   // If there is no element available, we wait.
   // We must check the condition again coming out
   // of the wait because of "barging" (use while instead of if)
   while (len == 0) {
      try{ wait();} catch (InterruptedException ignored) {}
   }
   // At this point, we have the lock and len>0
   System.out.println(Thread.currentThread().getName() + " remove " + (len-1));
   len--;
}

private class Adder extends Thread {
   public void run() {
      for (int i = 0; i< COUNT; i++) {
         add();
         yield(); // this just gets the threads to switch around more,
                  // so the output is a little more interesting
      }
   }
}

private class Remover extends Thread {
   public void run() {
      for (int i = 0; i< COUNT; i++) {
         remove();
         yield();
      }
      System.out.println(getName() + " done");
   }
}


public void demo () {

   // Make two "adding" threads
   Thread a1 = new Adder();
   Thread a2 = new Adder();



   // Make two "removing" threads
   Thread r1 = new Remover();
   Thread r2 = new Remover();

   // start them up (any order would work)
   a1.start();
   a2.start();
   r1.start();
   r2.start();

   /*
   output
      Add elem 0
      Add elem 1
      Remove elem 1
      Add elem 1
      Add elem 2
      Add elem 3
      Remove elem 3
      Remove elem 2
      Add elem 2
      Add elem 3
      Remove elem 3
      Remove elem 2
```

```
         Add elem 2
         ...
         Remove elem 3
         Remove elem 2
         done
         Remove elem 1
         Remove elem 0
         done
      */

   }

   public static void main(String[] args) {
       AddRemove test = new AddRemove();
       test.demo();
   }

}
```

Dropped Notify Problem

Notify does not "count" the number of notifies that happen

Instead, notify is instantaneous -- if there are waiters in the queue at the time of the notify, they will be signaled.

If a waiter comes to the queue after the notify, it is not signaled.

The classic CS "Semaphore" counts, so just realize that the wait/notify feature is more simple than a Semaphore.

It's possible to build a Semaphore on wait/notify

DroppedNotify

```
/**
 Demonstrates the "dropped notify" problem.
 Have one thread generate 10 notifies for use by another thread.
 Does not work: the notify works when the waiter is already in the queue, but
 a notify with no waiter does nothing.
*/
class DropNotify {
   // The shared point of contact between the two
   Object shared = new Object();

   // Collect 10 notifications on the shared object
   class Waiter extends Thread {
      public void run() {
         for (int i = 0; i<10; i++) {
            try {
               synchronized(shared) {
                  shared.wait();
                   System.out.println("wait");
               }
            } catch (InterruptedException ingored) {}
         }
         System.out.println("Waiter done");  // it never gets to this line
      }
   }

   // Do 10 notifications on the shared object
   class Notifier extends Thread {
      public void run() {
         for (int i = 0; i<10; i++) {
```

```
            synchronized(shared) {
              shared.notify();
              System.out.println("notify");
            }
         }
         System.out.println("Notifier done");
      }
   }


   public void demo() {
      new Waiter().start();
      new Notifier().start();
   }

   public static void main(String[] args) {
      new DropNotify().demo();
   }

    /*
    Output...
    notify
    notify
    notify
    wait
    notify
    notify
    notify
    notify
    notify
    notify
    notify
    Notifier done
    wait
    <hangs>
    ^C
    */

}
```

Semaphore Code

Here is some code to implement a counting Semaphore using Java lock/wait/notify primitives.
We are not going to semaphores at all in CS193j, but I'm including it for completeness.

```
/*
 The classic counting semaphore.
 You may want to save this code.
 Count>0 represents "available".
 Count==0 means the locks are all in use, and there may be
 threads waiting. The semaphore may be constructed with a negative count,
 A client should decr() to acquire, do work, and then incr() to release.
 If the semaphore was not available, decr() will block until it is.
 If a thread blocks in a decr() it is still holding all its other locks --
 Java's wait() does not automatically release locks othe than the receiver lock.

 In this version, decr() does not move the count < 0,
 so all incr() do a notify() -- they cannot know if
```

```
 a thread is waiting or not. This is ok, since a notify()
 with no waiting threads is cheap. In this case,
 a notify() is ok instead of a notifyAll(), since we know
 our count++ only enables a single waiter to run.
*/
class Semaphore {
   private int count;

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

   // Try to decrement. Block if count <=0.
   // Returns on success or interruption -- the caller
   // should check isInterrupted() if they care about that case.
   public synchronized void decr() {
      while  (count<=0) try {
         wait();
      }
      catch (InterruptedException inter) {
         // The exception does not set the "isInterrupted" boolean.
         // We set the boolean to true so our caller
         // will see that interruption has happened.
         Thread.currentThread().interrupt();
         return;
      }
      count--;
   }

   // Increase count by 1, potentially unblocking a waiter.
   public synchronized void incr() {
      count++;
      notify();
   }
}


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

   void a() {
      aGo.decr();
      System.out.println("It's A turn, A rules!");
      bGo.incr();
   }

   void b() {
      bGo.decr();
      System.out.println("It's B turn, B rules!");
      aGo.incr();
   }

   /*
    Q: Suppose a() and b() where synchronized -- what would happen?
    A: deadlock! a thread would take the "this" lock and block on its decr(),
    but the other thread could never get in to do the matching incr().
   */
   public void demo() {
       // Use this to wait for the workers
```

```
      final Semaphore finished = new Semaphore(-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();    // wait for both threads
      System.out.println("All done!");
   }

   public static void main(String args[]) {
      new TurnDemo2().demo();
   }
}
```

Swing/GUI Threading

Problem: Swing vs. Threads

How to integrate the Swing/GUI/drawing system with threads?

Problem: modifying the GUI state while, simultaneously, it is being drawn is not feasible -- a typical reader/writer conflict problem.

e.g. paintComponent() while another thread changes the component geometry

e.g. send mouseMoved() notification to an object, but another thread simultaneously deletes the object

Solution: Swing Thread aka One Big Lock

1. There is one, designated official "Swing thread"
2. The system does all Swing/GUI notifications using the Swing thread, one at a time..
    1. paintComponent() -- always on the Swing thread
    2. all notifications: action events, mouse motion events -- sent on the Swing thread
3. The system keeps a queue of "Swing jobs". When the swing thread is done with its current job, it gets the next one and does it.
4. Only the Swing thread is allowed to edit the state of the GUI -- the geometry, the nesting, the listeners, etc. of the Swing components.
5. Since only the Swing thread is allowed to touch the Swing related state, there is in effect a big lock over all the Swing state.

Programmer Rules...

1. On the swing thread -- edit ok

When you are on the swing thread, you are allowed to edit the swing state.

e.g. container.add(), setPreferredSize(), setLayout()

2. Don't hog the swing thread

Do not do time-consuming operations on the swing thread

There is only one swing thread, and you have it. No Swing/GUI event processing will happen until you finish your processing and return the swing thread to the system -- then it can dequeue mouse events, push in buttons, etc.

Fork off a worker thread to do a time-consuming operation

3. Not on the swing thread -- no edit

A thread which is not the swing thread may not send messages that edit the swing state (add(), setBounds(), ...).

Use invokeLater (below) to run code on the swing thread

One exception is repaint(), any thread may send repaint(). The other allowed messages are revalidate(), and add/remove Listener messages.

Another exception is before the component has been brought on screen (setVisible(true))
-- before the component is on screen, it's ok to add(), setBounds(), etc. on it.

Swing Thread: Results

1. In your notifications (paintComponent(), actionPerformed()) -- you are on the Swing thread. Feel free to send swing messages.
2. There is only one swing thread, so when you have it, no other Swing thread activity is happening. No other thread is drawing or changing the Swing state, geometry, etc.

SwingUtilities

Built in utility methods that allow you to "post" some code to the swing thread to run later.

"Runnable" interface -- defines public void run() { }

SwingUtilities.invokeLater(Runnable)

Queue up the given runnable -- the swing thread will execute the runnable when the swing thread gets to it in the queue of things to do.

SwingUtilities.invokeAndWait(Runnable)

As above, but block the current thread (a worker thread of some sort presumably), until the runnable exits.

SwingUtilities Client Example

Suppose we have a typical MyFrame class that contains a JLabel. There is a worker thread that wants to modify the label text.

Put together a Runnable inner class on the fly to do it -- like a lambda or function pointer.

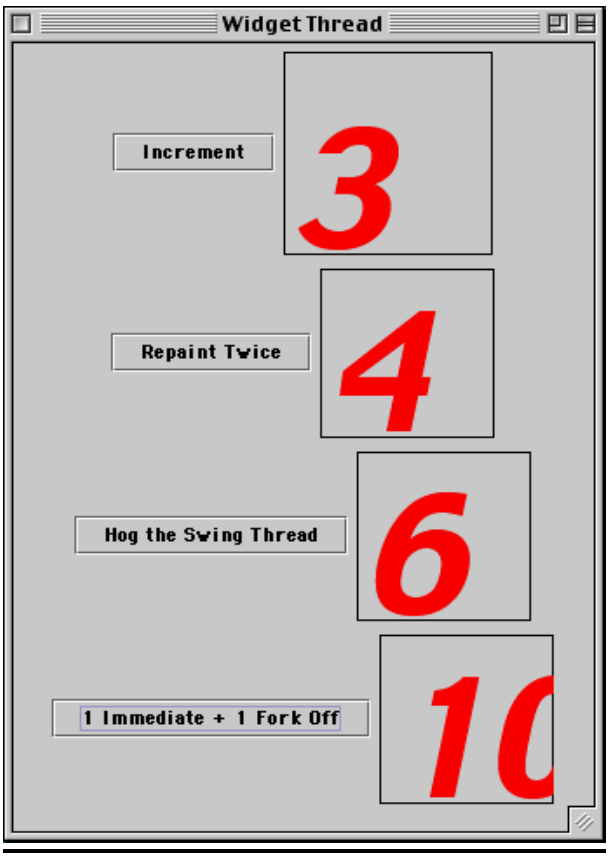
```
class MyFrame extends JFrame {
   private JLabel label;

   // Typical GUI code down here creates and starts the worker
   public MyFrame() {
      // standard Frame ctor stuff, create buttons...
      button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Worker worker = new Worker();
            worker.start();
         }
      });
      ...
   }


   class Worker extends Thread {
      public void run() {
         // The worker does some big computation
         final String answer = <something>;

         // We want to call setText() to send the answer to the GUI.
         // We must go through SwingUtilities.invokeLater()
         SwingUtilities.invokeLater(
            new Runnable() {  // create a runnable on the fly
               public void run() {
                  label.setText(answer);
               }
            }
         );
      }
   }


}
```

SwingThread Demo

Demonstrates swing thread issues

-Hogging the swing thread -- bad!

-Fork off a worker + worker uses SwingUtilities.invokeLater() to communicate back to the swing state



SwingThread Code

```
// SwingThread.java
/*
 Demonstrates the role of the swing thread and how
 to fork off a worker thread that uses invokeLater().

 The Widget class represents a typical Swing object.
 A Widget draws an int value and responds to the increment() message.
 The increment() message should only be sent on the Swing thread.
*/
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;


public class SwingThread extends JFrame {
    private Widget a, b, c, d;
    public final int SIZE = 75;

    public SwingThread() {
        super("Swing Thread");


      JComponent content = (JComponent) getContentPane();
      content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));

      // 1. Simple
      // Just call increment() -- fine, we're on the swing thread
```

```
// so we are allowed to message and change swing state.
// Result: works fine
a = new Widget(SIZE, SIZE);
JButton button = new JButton("Increment");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      a.increment();
   }
});

JPanel panel = new JPanel();
panel.add(button);
panel.add(a);
content.add(panel);


// 2. Coalesing
// Extra call to repaint() -- first of all, it's not necessary
// since increment() calls repaint(). Second of all, the two repaints
// are coalesced into a single draw operation anyway.
// The Swing thread only has a chance to do anything about it after
// our actionPerformed() exits.
// Result: the extra repaint() is basically harmless, so works fine
b = new Widget(SIZE, SIZE);
button = new JButton("Repaint Twice");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      b.increment();
      b.repaint();
   }
});

panel = new JPanel();
panel.add(button);
panel.add(b);
content.add(panel);



// 3. Hog The Swing thread
// Notice how the UI locks up while we hog the swing thread.
// Also, the widget _never_ shows an odd number. The draw thread
// only gets a chance to do anything after we exit actionPerformed()
// and by then we've always incremented to an even number.
// Result: button goes in, stays in for a few seconds, UI locks up,
// then button pops out and widget draws with +2 value
// Lesson: don't hog the swing thread
c = new Widget(SIZE, SIZE);
button = new JButton("Hog the Swing Thread");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      c.increment();
      int sum = 0;
      for (int i=0; i<120000000; i++) { sum += i; }
      c.increment();
   }
});

panel = new JPanel();
panel.add(button);
panel.add(c);
content.add(panel);
```

```
      // 4. Here we increment once immediately,
      // and then fork off a worker to do something time-consuming
      // and then increment when it is done. Notice that the UI
      // remains responsive while the worker is off doing its thing.
      // The worker uses SwingUtilities.invokeLater() to communicate
      // back to swing.
      // Result: button goes in and out normally, one increment happens
      // immediately, UI remains responsive, after a few seconds, the
      // widget increments a second time on its own. It is possible to
      // click the button multiple times to fork off multiple, concurrent
      // workers, or could use an interruption strategy on the previous worker.
      // Q: is there a possible writer/writer conflict with multiple
      // workers calling increment() at the same time?
      d = new Widget(SIZE, SIZE);
      button = new JButton("1 Immediate + 1 Fork Off");
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            // 1. Do an increment right away
            // -- ok, we're on the swing thread
            d.increment();

            // 2. Fork off a worker to do the iterations
            // on its own, followed by the increment()
            Thread worker = new Thread() {
               public void run() {
                    int sum = 0;
                    for (int i=0; i<120000000; i++) { sum += i; }

                  // 3. When worker wants to communicate back to swing,
                  // must go through invokeLater/runnable
                  SwingUtilities.invokeLater(
                     new Runnable() {
                        public void run() {
                           d.increment();
                        }
                     }
                  );

               }
            };
            worker.start();
         }
      });
      panel = new JPanel();
      panel.add(button);
      panel.add(d);
      content.add(panel);


      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pack();
      setVisible(true);

   }
}
```

Q: Repaint Concurrency Problem?

Here's the standard looking code to deal with move/repaint used in the dot example...

```
   public void movePoint(Point point, int dx, int dy) {
```

```
        repaintPoint(point.x, point.y);  // old

        point.x += dx;
        point.y += dy;

        repaintPoint(point.x, point.y);  // new
}
```

Q: Is there a race condition problem where the first repaint causes a paintComponent() to come through before the point.x += dx goes through? In that case, the "erase" would not happen correctly.

A: No problem -- one big lock

We are on the swing thread since we are in a mouse notification (all swing notifications are on the swing thread)

The paintComponent needs the swing thread, so it won't happen until we exit out mouse listener releasing the swing thread back to the system. Until then, the paint is waiting in the swing queue.

It's like there's one big lock, and we have it. This gives us the luxury of not worrying about concurrency problems mostly.

Threading Conclusions

Java -- OOP Concurrency Style

The simple case for java is: objects store state, getters/setters for that state are declared synchronized, and so access against that state is thread state.

An intuitive extension of the OOP idea to threading -- not just a translation of C/C++ style lock()/unlock() ideas.

CT "Structured" Style

Notice in java, the lock/unlock structure is specified at compile-time

e.g. synchronized(obj) { ....<statements> .... }

It is not possible to write code where the lock/unlock does not balance -- since it's structured at compile time

Contrast to lock(obj); ..... unlock(obj) systems

Historically, what Java does is called "monitor" style locking

Tradeoff: not completely flexible, but makes coding for many cases more simple -- CT structure and error checking.

Single Threaded is Easier

Writing single-threaded code is much easier.

There will always be cases where writing the single threaded version is the best use of time.

In some circumstances, it may be worth the extra effort to build the multi-threaded version.

When to Thread

1. Hardware

Someday, threading may be an important program feature to boost performance as hardware moves towards increased thread parallelism. Increasingly parallel hardware may make this option more important in the future.

2. GUI

Fork off worker threads so that the GUI remains responsive. Use SwingUtilities.invokeLater(...) to communicate back to the swing thread. A good usability feature if the app has some operations which are pretty slow. Works even with only one processor.

3. Networking

Use threads to support multiple connections. Great speedup, since it allows you to overlap ("pipeline") the multiple, relatively slow network operations. Works well, even if you only have one processor.

Design for Concurrency

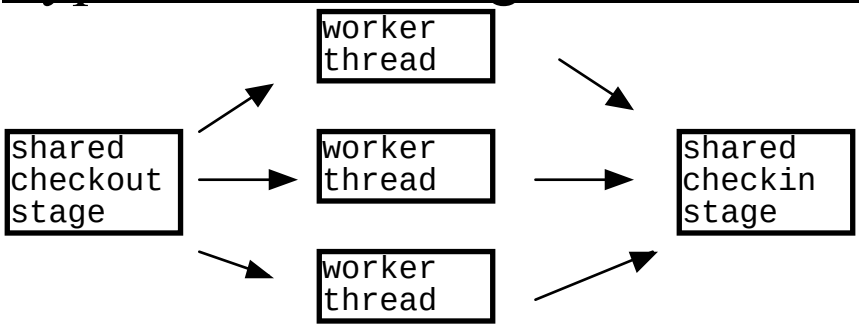
By default, do not put much effort in to making a class support concurrency.

Like subclassing, support for concurrency should be deliberately added to a class only in the cases where it really makes sense. Adding support for concurrency is not a trivial little operation.

Does the pattern of locks allow multiple threads to get useful work done at the same time?

There will be some moments when threads must wait for each other, but ideally, much of the time they can each proceed independently.

Typical Good Design -- Checkin/Checkout



There are many worker objects. Each "checks out" data from a shared input stage, but is then able to proceed independently for the most part -- not accessing (locking) shared memory. When it's finished, the worker typically needs to "check in" its results to some sort of shared output stage. Parallelism will work best if the checkin/checkout operation is small compared to the work each worker can do.

e.g. SETI

Bad design: Each thread frequently accesses a shared (locked) object, that the other threads are also accessing -- there is so much "contention" that the workers get in each other's way.

N Processors -- Not N times Faster

A concurrent version of a program is not necessarily N times faster on an N processor machine. It is very common that the threads interfere with each other enough to slow things down significantly from the theoretical peak. Of course it depends on how independent the worker threads can be from each other once forked. The SETI program, for example, gets almost perfect linear speedup, since the checkout/checking phases are so small relative to the independent work phase. Most real-world problems are not that cleanly concurrent.