Threading 3

Previous handout

synchronized methods
synchronized(obj) { ...} form / coarse vs. fine locks
Get In Get Out
    Hold the lock as little as possible
    Do work outside the lock-hold region
sleep() -- current running thread
yield() -- current running thread

Interrupt

(previous handout)
vs. stop()
1. Other thread sends worker.interrupt() --sets the interrupted bit
2. Worker checks isInterrupted() occasionally
3. -or- Worker gets InterruptedException out of a sleep() or wait()
4. Worker exits its run() cleanly, leaving data in some consistent state

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
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..
    paintComponent()
    all notifications: action events, mouse motion events
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. In effect, there is one lock for all of the swing state -- the swing thread has it, and no other thread ever has it.

Swing Thread: Results

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

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(), ...).
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 -- before the component is on screen, it's ok to add(), setBounds(), etc. on it.

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 Code

Suppose we have a typical MyFrame class that contains a JPanel. There is a worker thread that wants to modify the panel when it starts -- adding a "worker started" JLabel.

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

```
class MyFrame extends JFrame {
   private JPanel panel;

   class Worker extends Thread {
      public void run() {

         // we want to add a JLabel saying that we have started.
         // Use SwingUtilities
         SwingUtilities.invokeLater(
            new Runnable() {  // create a runnable on the fly
               public void run() {
                  panel.add(new JLabel("worker started");
               }
            }
         );
         // do whatever else the worker run() does
         ...
      }
   }

   // 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();
         }
      });
      ...
   }
}
```

Worker vs. Swing -- Two Strategies

1. Use SwingUtilities.invokeLater
   - Syntax is a bit messy
   - Very simple for concurrency -- no conflicts, since there is only one swing thread, each thread operates in its own turn.
   - This is our preferred solution
2. Synchronization
   - It's possible to use classic synchronization so the worker and swing threads can use the same data.
   - Worker threads can still send repaint() to get the swing thread to look at the new data.Worker vs. GUI -- 2 solutions

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.

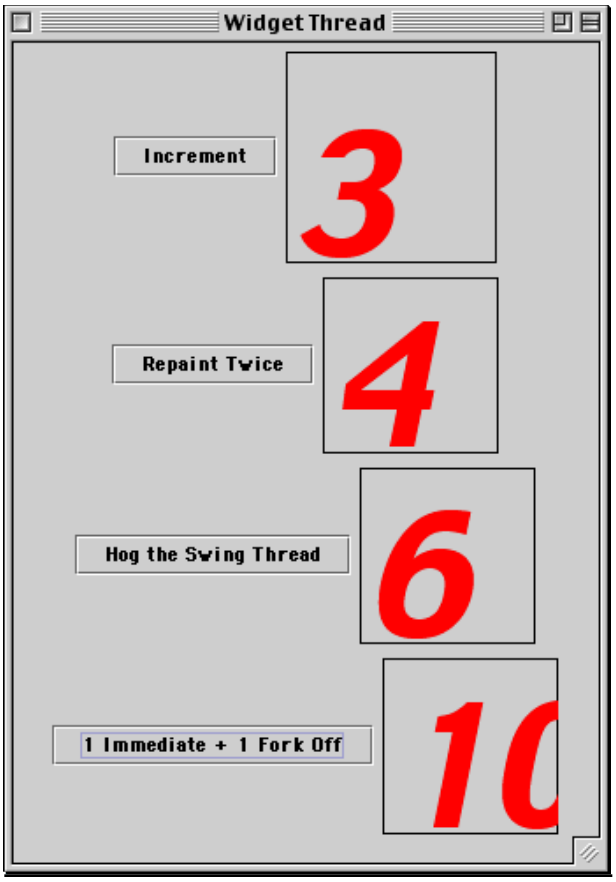
It's like there's one big lock, and we have it. This gives us the luxury of not worrying about concurrency problems mostly.

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



WidgetThread Code

```
// SwingThread.java
/*
 Demonstrates the role of the swing thread and how
 to fork off a worker thread.

 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  {


   public static void main(String[] args) {
      JFrame frame = new JFrame("Swing Thread");
      JComponent content = (JComponent) frame.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
final Widget a = new Widget(120, 120);
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.
// Result: the extra repain() is basically harmless, so works fine
final Widget b = new Widget(100, 100);
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 have incremented
// the widget 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
final Widget c = new Widget(100, 100);
button = new JButton("Hog the Swing Thread");
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      c.increment();
      for (int i=0; i<120000000; 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?
      final Widget d = new Widget(100, 100);
      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() {
                  for (int i=0; i<120000000; 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);

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

   }

}
```

Few vs. Many Locks (Coarse vs Fine)

Tradeoffs for few / coarse locks vs many / fine locks
Many locks: allow more concurrency between threads, more complex, more time spent locking / unlocking
Few Locks: allows less concurrency, simpler design, less time spent locking / unlocking
Conclusion: use few locks at first, re-design if concurrency is too restricted

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 your 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.

Advanced Java Threading

Wait / Notify -- used to block efficiently + allow threads to signal each other
Our "try again in 200 milliseconds" strategy in the HW3 Thread Bank problem is a little sloppy -- the best solution uses efficient blocking with wait / notify.
Semaphore(int) -- can build a classical counting semaphore with the Java synchronized / wait / notify primitives.

Java Thread Conclusions

Object based synchronization -- a pretty simple locking style
Future: hardware support for many threads
1. Someday, threading may be an important program feature to boost performance as hardware moves towards increased thread parallelism.
2. Networking -- use threads to support multiple connections. Works well, even if you only have one processor.
3. GUI -- fork off worker threads so that the GUI remains responsive. Use SwingUtilities.invokeLater(...) to communicate back to the swing thread.