Lecture 36

Threads 2

Reading for next time: Big Java 4.2, 22.1-22.6

Synchronization of Threads

Once your programs use threads, you often must deal with the conflicts and inconsistencies threads can cause.

The two most significant problems are *synchronization* and *deadlock*.

Synchronization, the Problem

- In many situations a segment of code must be executed either "all or nothing" before another thread can execute.
- For example, suppose you are inserting a new object into an `ArrayList` and the new item exceeds the current capacity. The `ArrayList` method `add()` will need to copy the `ArrayList` contents to a new piece of memory with greater capacity and then add the new element.
- If this operation is being executed by one thread and is partially completed when another thread gets control and attempts to get an element from the same `ArrayList`, we have a problem. The interrupted first thread will have left the partially copied vector in an inconsistent state.
- (`ArrayList` is synchronized to avoid this problem!)

`synchronized` Methods

- Java allows you to declare a method as `synchronized` to avoid such problems.
- A method definition such as

```
public synchronized void compute() {
  // body of method
}
```

  means that `compute()` can not be interrupted by another `synchronized` method acting on the same object.
- If another thread attempts to execute another `synchronized` method on the same object, this thread will wait until the first `synchronized` method exits.

`synchronized` Method Cautions

- But note that `synchronized` methods only wait for other `synchronized` methods.
- Normal, unsynchronized methods invoked on the same object will proceed.
  - That's usually ok. Unsynchronized methods often only read data and don't write it, or don't look at the data that might be affected by synchronization
- And another thread can run another `synchronized` method on another instance (object) of the same class.
  - That's usually ok. No data errors will occur; different objects are being used. If your different objects manipulate a common vector, for example, there may still be a problem.

How Synchronization Works

- Java implements `synchronized` methods via a special lock called a monitor that is a part of every instance of every class that inherits from `Object`.
- When a thread needs to enter a `synchronized` method, it tries to acquire the lock on the current object.
- If no other `synchronized` method called on this object is in progress in any thread, then the lock is free and the thread can proceed. But if another thread is executing a `synchronized` method on the object, then the lock will not be free and the first method must wait.
- If a static method is `synchronized`, then the lock is part of the object representing the class

Synchronization in the JDK

- The trick is knowing when a method needs to be synchronized. Many methods in the predefined Java classes are already synchronized.
- For instance, most methods of the `ArrayList` class, are synchronized for the reason we pointed out above.
- As another example, the method of the Java AWT `Component` class that adds a `MouseListener` object to a `Component` (so that `MouseEvents` are reported to the `MouseListener`) is also synchronized. If you check the AWT and Swing source code, you find that the signature of this method is

```
public synchronized void
   addMouseListener(MouseListener l)
```

Java Synchronization Defaults

- By default, (i.e. unless you declare otherwise), methods are NOT `synchronized`.
- Declaring a method `synchronized` slows down the execution of your program because acquiring and releasing the locks generates overhead.
- It also introduces the possibility of a new type of failure called deadlock
- However, in many cases it is essential to synchronize methods for your program to run correctly.

Lossless Router Example

```
public class RouterTest {
   public static final int PORTS= 50;
   public static final int BUFFER_SIZE= 10000;

   public static void main(String[] args) {
     Router r= new Router(PORTS);
     for (int i=0; i < PORTS; i++) {
       TransferThread t= new TransferThread(r, i, BUFFER_SIZE);
       if (i > PORTS/2)
          t.setPriority( Thread.MIN_PRIORITY);
       else
          t.setPriority( Thread.MAX_PRIORITY);
       t.start();
     }
   }
}  // Safer to have array of Threads. Non-Java threads packages
   // could consider threads out of scope when t is reused
```

Lossless Router Example, 2

```
public class Router {
   private int[] port;
   private long packets= 0;
   public Router(int n) {
        port= new int[n];
   }
   public void transfer(int from, int to, int bytes) {
        port[from] -= bytes;
        port[to] += bytes;
        packets++;
        if (packets % 10000 == 0)
               test();
   }
   public void test() {
        int sum= 0;
        for (int i= 0; i < port.length; i++)
               sum += port[i];
        System.out.println("Packets: "+packets+" net: "+ sum);

   }
}
```

Lossless Router Example, 3

```
public class TransferThread extends Thread {
   private Router router;
   private int fromPort;
   private int maxBytes;

   public TransferThread(Router r, int f, int max) {
     router= r;
     fromPort= f;
     maxBytes= max;
   }
   public void run() {
     try {
        while (!interrupted()) {
          int toPort= (int) (RouterTest.PORTS * Math.random());
          int bytesSent= (int) (maxBytes * Math.random());
          router.transfer(fromPort, toPort, bytesSent);
          sleep(1);
        }
     } catch (InterruptedException e) {
     }
   }
}
```

Exercise

- **Download RouterTest, Router, TransferThread and run RouterTest**
  - **For a long time, if necessary**
    - **Go to Debug perspective and select 'Terminate' to end the program**
  - **Is your router lossless?**
  - **If not, what's going on?**
    - **Router is truly lossless**
    - **transfer() method is correct**

In case it doesn't misbehave…

```
Packets: 10000 sum: 0
Packets: 20000 sum: 0
Packets: 30000 sum: 0
…
Packets: 630000 sum: 0
Packets: 640000 sum: 0
Packets: 650000 sum: 3281
Packets: 660000 sum: 3281
Packets: 670000 sum: 3281
…
Packets: 1190000 sum: 3281
Packets: 1200026 sum: 3281
Packets: 1210000 sum: -19099
Packets: 1220000 sum: -19099
…
```

Solution

- Is your router lossless?
  - Yes, the hardware really is!
- If not, what's going on?
  - A synchronization problem

```
public void transfer(int from, int to, int bytes) {
        port[from] -= bytes;
        port[to] += bytes;
        packets++;
        if (packets % 10000 == 0)
                test();
}
```

  - A thread can be interrupted by another thread between writing port[from] and port[to], or even during each of those statements. If the other thread gets in between these two operations, the result will be wrong

Solution, 2

- For example: Initial conditions
  - Port 1: +500, Port 2: -300
  - Thread A sends 50 bytes from port 1 to port 2
    - Decrements port 1 by 50: +450
  - Thread B interrupts and sends 100 bytes from port 2 to port 1
    - Decrements port 2 by 100: -400
    - Increments port 1 by 100: +600 or +550, depending on timing!
  - Thread A then completes
    - Increments port 2 by 50, from –300 to –250
    - It overwrites thread B's result because it had read the current port 1 and 2 byte counts from memory before updating them (and before thread B updated them)
  - This is a 'race condition': first one loses, tho!

Lossless Router

Port 1	Port 2	Thread A	Thread B
+500	-300	Gets (+500, -300) Moves 50 bytes 1->2	
+450		Updates port 1 Interrupted by B Has not updated port 2	
+550	-400		Gets (+450,-300) Moves 100 bytes 2->1 Updates ports 1 and 2
	-250	Thread A completes Updates port 2, overwrites B's update	
50 bytes gained	50 bytes gained	Thread B's decrement of 100 bytes in port 2 lost!	

Exercise, continued

- **Fix the synchronization problem**
  - **Use the synchronized keyword**
    - **It must follow the public keyword and precede the return type keyword**
  - **Figure out where to use it**
  - **When you've done it, compile and run it**
    - **See if the net bytes ever stray from zero**

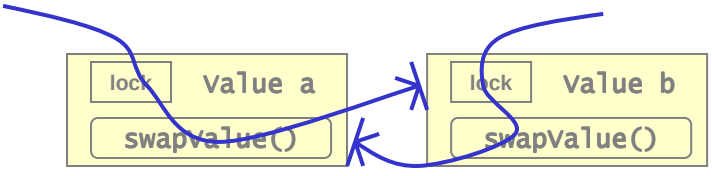
Deadlock

- **When two different threads each require exclusive access to the same resources, you can have situations where each gets access to one of the resources the other thread needs. Neither thread can proceed.**
- **For example, suppose each of two threads needs exclusive privilege to write two different files. Thread 1 could open file A exclusively, and Thread 2 could open file B exclusively.**
- **Now Thread 1 needs exclusive access to File B, and Thread 2 needs exclusive access to file A. Both threads are stymied. The most common source of this problem occurs when two threads attempt to run `synchronized` methods on the same set of objects.**

Deadlock Example

```
public class Value
{
  private long value;
  public Value( long v ) { value=v; }
  synchronized long getValue() { return value; }
  synchronized void setValue( long v) { value=v; }
  synchronized void swapValue( Value other ) {
    long t = getValue();
    long v = other.getValue();
    setValue( v );
    other.setValue(t);
  }
}   // This is conceptual. There are some
    // picky details to simulate deadlock that we don't
    // address. They happen in real life by themselves…
```

Deadlock Diagram



After Doug Lea, *Concurrent Programming in Java* (2000), excellent but advanced reference

The Symptoms of Deadlock

- **The symptoms of deadlock are that a program simply hangs (stops executing) or that a portion of the program governed by a particular thread is endlessly postponed.**
- **Synchronization and deadlock problems are miserably hard to debug because a program with such problems may run correctly many times before it fails.**
- **This happens because the order and timing of different `Threads`' execution isn't entirely predictable.**
- **Programs need to be correct independent of the order and timing with which different Threads are executed.**
- **As soon as you synchronize in order to prevent harmful interference between threads, you risk deadlock.**

Threads and Swing

All Java programs run at least three threads:

1. **the `main()` thread; that is, the thread that begins with your main method;**
2. **the event thread, on which the windowing system notifies you of the events for which you have registered; and,**
3. **the garbage collection thread.**

- **The garbage collection thread runs in the background (at a low priority), and you can usually forget that it is there.**
- **But as soon as you put up a graphic user interface, you have to take account of the event thread.**

Threads and the AWT

- **The initial Java GUI package, the AWT, synchronized many methods in the GUI classes. But it made the AWT classes susceptible to deadlock.**
- **When the Java programmers set out to implement the vastly more complex capabilities of Swing, they, in effect, gave up.**
- **The AWT attempts to be multithreaded, that is, to allow calls from multiple threads.**

Threads and Swing

- **With a very few exceptions, the Swing classes expect to have their methods called only from the event thread. As the Java developers state it:**

  "*Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread."*

Threads and Swing, 2

- A component is *realized* when the windowing system associates it with a window that will actually paint it on the screen.
- Usually this happens when the component is first made visible or when it is first given an accurate size (via a call to pack(), for instance).
- Up until then it can be modified from another thread like the main thread because there is no chance that it will be accessed from the event thread until the windowing system knows about it.
- So you can `add()` components to a container from the main thread or add text to a `JTextArea`, as long as it is not realized.

Threads and Swing, 3

- But once it has become visible, it can receive mouse clicks or key presses or any other type of event, and the corresponding listener methods may be used.
- Swing does NOT synchronize these methods or the methods that they may call such as `setText()` or `add()`.
- If you want to call `setText()` or methods like it from any other thread than the event thread, you should use a special technique.

Modifying a GUI from Another Thread

- Essentially, you create an object that describes a task to be performed in the event thread at some future time.
- Then you pass that task to the event thread using a synchronized method that queues it up with the other events in the event thread's event queue.
- Swing will execute the task when it wants, but because Swing only processes one event at a time including these special tasks, they may call unsynchronized methods on the GUI classes.

Using `invokeLater()`

- How do we create such a task?

```
Runnable update = new Runnable() {
  public void run() {
    component.doSomething();
};
SwingUtilities.invokeLater( update );
```

- `invokeLater()` is a synchronized static method in the `SwingUtilities` class in the `javax.swing` package. It inserts the task in the event queue.

Synchronized Swing Methods

There are some Swing methods that may safely be called from another thread. These include:

- `public void repaint()`
- `public void revalidate()`
- `public void add`*`EventType`*`Listener(Listener l)`
- `public void remove`*`EventType`*`Listener(Listener l)`

JFileViewer

```
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class JFileViewer extends JFrame {
   private static JFileViewer view;
   private JTextArea text;

   public JFileViewer(String path) {
        super(path);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        text = new JTextArea(20, 60);
        text.setLineWrap(true);
        JScrollPane p= new JScrollPane(text, …) // See code
        Container c= getContentPane();
        c.add(p, BorderLayout.CENTER);
        pack();
        setVisible(true);
   }
   public void append(String s) {
        text.append(s);
   }
```

JFileViewer, 2

```
    public static void main(String[] args) {
        String filename = "C:/Test.txt";
        view = new JFileViewer(filename);
        try {
                FileReader in = new FileReader(filename);
                BufferedReader b = new BufferedReader(in);
                String s;
                while ((s = b.readLine()) != null) {
                        view.append(s + "\n");
                }
                in.close();
        } catch (IOException e) {
                System.err.println(e);
                view.setVisible(false);
                view.dispose();
                System.exit(1);
        }
    }
}
```

Exercise

- **Download JBetterFileViewer**
  - **This displays the scroll pane first and then adds the text file, line by line.**
  - **It must notify the scroll pane when it has read another line. To do this it must create a Runnable object when it has a new line of text and then call invokeLater() to put the object on the Swing event list**
- **Exercise**
  - **Complete the main() method**
  - **Refer to the "Using invokeLater()" slide for what to do**
    - **You are creating an object of type Runnable that you will place on the Swing event list by calling invokeLater with the object as its argument**
    - **The object will be an anonymous inner class. It must have a run() method to implement the Runnable interface**

JBetterFileViewer

```
import java.io.*;                    // Identical to page 1
import javax.swing.*;                // of JFileViewer
import java.awt.*;

public class JBetterFileViewer extends JFrame {
   private static JBetterFileViewer view;
   private JTextArea text;

   public JBetterFileViewer(String path) {
        super(path);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        text = new JTextArea(20, 60);
        text.setLineWrap(true);
        JScrollPane p= new JScrollPane(text, …) // See code
        Container c= getContentPane();
        c.add(p, BorderLayout.CENTER);
        pack();
        setVisible(true);
   }
   public void append(String s) {
        text.append(s);
   }
```

JBetterFileViewer, 2

```
   public static void main(String[] args) {
     String filename = "C:/Test.txt";
     view = new JBetterFileViewer(filename);
     try {
        FileReader in = new FileReader(filename);
        int nread;
        char [] buf = new char[ 512 ];
        while( ( nread = in.read( buf ) ) >= 0 ) {
            final String s= new String(buf, 0, nread);
            // Your code here:
            // Create an object of type Runnable
            //   Write its run() method
            // Call invokeLater()
          }
          in.close();
        }
        catch (IOException e) {
          System.err.println(e);
          view.setVisible(false);
          view.dispose();
          System.exit(1);
} } }
```

Common Sense Rules for `Threads`

1. **Only use multiple `Threads` when they are essential:**
   - **Multiple streams, multiple computations and it's too slow to do them in sequence**
2. **Decide whether the methods you wrote may need to be `synchronized`:**
   - **Are they altering common resources across threads?**
   - **When in doubt, declare methods as `synchronized`.**
3. **Make sure `Threads` die off as soon as they aren't needed.**
4. **If different runs of the same program with more than one thread execute differently even though they are given the same inputs, you probably have a synchronization problem**
   - **These can be very hard to find and correct**