1.00 Lecture 29

Sensors and Threads

Reading for next time: Numerical Recipes 32-36 (online)
http://www.nrbook.com/a/bookcpdf.php

Threads and Sensors

- When sensors send events to a Java program, a `SensorChangeEvent` is delivered to a Java program on a separate thread.
- Class InterfaceKitPhidget uses 3 threads:
  - Central thread that looks for AttachEvents and DetachEvents
  - Read thread that gets `SensorChangeEvents`
  - Write thread that manages `setOutputState()`
- All Phidgets libraries are thread safe, as we'll discuss later in this lecture
- Event listeners are used to get events
  - Listeners are registered with the Phidgets library

Threads and Atomic Operations

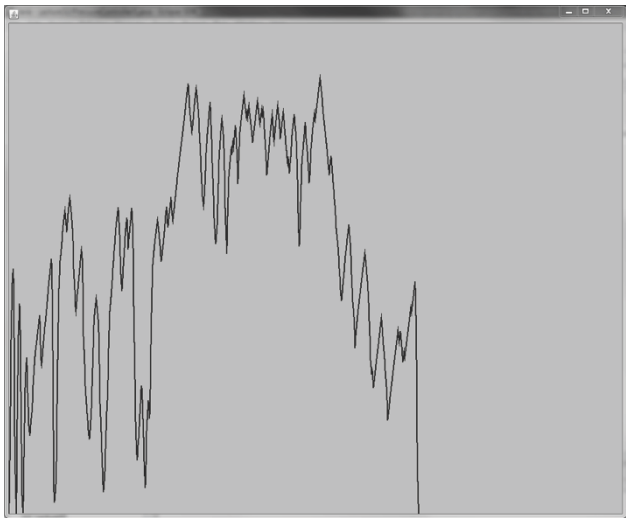
- If two threads access the same data, these operations must be atomic, or execute without interruption by another thread accessing the same resource. Why? For example,
  - Thread inserts a new object into an `ArrayList` and the new item exceeds the current capacity.
  - `ArrayList` method `add()` must copy `ArrayList` contents to a new piece of memory with greater capacity and then add the new element.
  - If add() 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`, the interrupted first thread will have left the partially copied list in an inconsistent state.
  - The result is unpredictable and may be garbage

`PressureController4` Program

- Download and run `PressureController4` and `PressureView4`
- On most computers this program will fail after a period of time because it does not share a critical resource atomically
  - ConcurrentModificationException is thrown: conflict in ArrayList
- Threads execute probabilistically. The program will run for a different length of time on each execution.
  - On some systems, it may not fail at all.
- `PressureController4` is almost identical to earlier versions except that it registers for `SensorChangeEvents` as follows:

```
interfaceKit.addSensorChangeListener(new
   SensorChangeListener() {
     public void sensorChanged(SensorChangeEvent se) {
       if ( se.getIndex() == pressureIndex )
         pv.updateChart( se.getValue() );  // Full history
}});
// Earlier versions just set pressure and repainted the
// one data point. This version shows the full history.
```

PressureController4 sample output



PressureView4

- The data for the pressure graph is stored in

```
private ArrayList<Integer> vals;
```

- The `updateChart()` method is called from `PressureController4` whenever a `SensorChangeEvent` is received.

```
public void updateChart(int newVal) {
  vals.add(newVal);     // Writes to ArrayList
  repaint();            // Reads from ArrayList
}
```

- But `paintComponent()`, triggered by `repaint()`, uses the same `ArrayList` to draw the graph
- The Phidget objects create and start threads
- Swing creates and starts a Swing thread
- Thus, a Phidget thread and the Swing thread contend for the ArrayList

Why No Problem in past `PressureController`?

- In past version, the `SensorChangeEvent` gets pressure value:

```
private void updateSensor(SensorChangeEvent se) {
  System.out.println(se);
  if (se.getIndex() == pressureIndex) {
    pressure = se.getValue();
    pv.repaint();             // PressureView
}}
```

- `PressureView's repaint()` calls `getPressure()` from `PressureController`, so the value of pressure is shared between a Phidgets thread and a Swing thread

```
public void paintComponent( Graphics g ) { …
 double x= 100;
 double height=((double)model.getPressure()/1000.0)* 300;
```

- But Java variable assignment (4 bytes or less) is guaranteed to be atomic. It cannot be interrupted when partially complete.
  - An ArrayList can be interrupted, in PressureController4
  - We used a float (4 bytes) in CylinderThread rather than a double (8 bytes) for this reason in the last lecture. Java is a bit behind the times on this one…

`synchronized` Methods

- Java has `synchronized` keyword to avoid such problems:

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

- `compute()` cannot be interrupted by another `synchronized` method acting on the same object.
- If a second thread attempts to execute another `synchronized` method on the same object, the second thread will wait until the first `synchronized` method exits.

Exercise 1: `PressureController4`

- Use the `synchronized` keyword to fix the `PressureController4` program and test it.
- What methods need to be synchronized in `PressureController4` and `PressureView4`?
  - Use the smallest scope possible for the synchronized keyword. Focus on where the ArrayList is used.
  - The Swing thread and a Phidgets thread are the two that interact over the ArrayList.
- You should not have to change any code other than the method declarations.

Example 2: Life without Synchronization

Imagine: You and your mother deposit $100 into your account. Take a moment to understand the code. (More details in download)

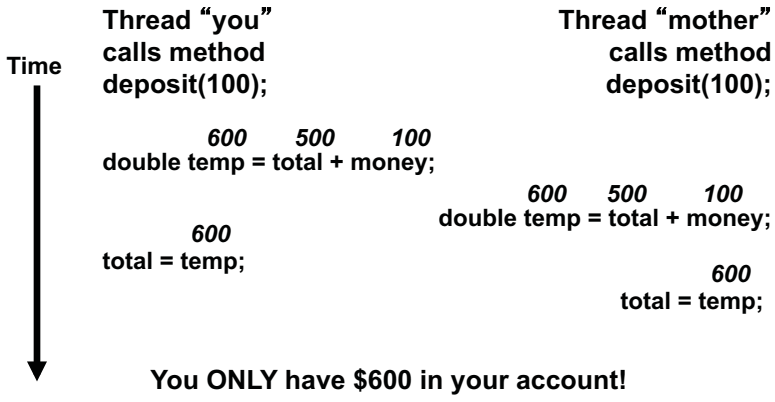
```
public class BankAccount{
   double total = 0.0;
   public BankAccount(double total){
        this.total = total;
   }
   public void deposit ( double money ){
        double temp = total + money;
        total = temp;
   }
}
```

```
public class ATM implements Runnable{
   private BankAccount ba;
   public ATM(BankAccount ba){  this.ba = ba; }
   public void run(){  ba.deposit(100);  }
   public static void main(String[] args){
        BankAccount david = new BankAccount(500);
        Thread you = new Thread(new ATM(david));
        Thread mother = new Thread(new ATM(david));
   }    // Start threads, join, etc.–see download
}
```

Life without Synchronization, p.2

Imagine: You AND your mother deposit $100 into your account.

What if the code gets scheduled to run as follows?



You ONLY have $600 in your account!

Exercise 2: Synchronization

- **Run ATM to see the problem.**
- **Correct ATM.**
- **Compile and run it a few times to make sure that the problem is gone.**

`synchronized` Method Cautions

- `synchronized` methods only wait for other `synchronized` methods.
- Normal, unsynchronized methods invoked on the same object will proceed.
  - Unsynchronized methods that don't use the data that might be affected by synchronization are ok
- Another thread can run another `synchronized` method on a different 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 ArrayList, for example, there may still be a problem.
- By default, methods are NOT `synchronized`.

How Synchronization Works

- Java implements `synchronized` methods via a lock on the object being accessed
- When a thread calls a `synchronized` method, it tries to acquire the lock on the 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.
- If another thread is executing a `synchronized` method on the object, then the lock will not be free and the first method must wait.
- A library (e.g., the Phidget library) is thread-safe if all methods that might interfere with each other are synchronized.

Why not synchronize always?

- When two different threads each require exclusive access to the same resources, situations occur where each gets access to one of the resources the other thread needs. Neither thread can proceed.
- Suppose each of two threads needs exclusive privilege to write two different files.
  - Thread 1 opens file A exclusively
  - Thread 2 opens file B exclusively
- Now Thread 1 also needs exclusive access to file B, and Thread 2 also needs exclusive access to file A. Both threads are deadlocked.
- The most common source of this problem occurs when two threads attempt to run `synchronized` methods on the same set of objects.

Symptoms of Deadlock

- The symptoms of deadlock are:
  - Program hangs (stops executing), or
  - Portion of program governed by a particular thread is endlessly postponed.
- Synchronization and deadlock problems are hard to debug because a program with such problems may run correctly many times before it fails.
  - Order and timing of different `Threads`' execution isn't entirely predictable.
- Programs must be correct independent of the order and timing that its Threads are executed.
- If you synchronize in order to prevent harmful interference between threads, you risk deadlock.

Threads and Swing

- With a very few exceptions, the Swing classes expect to have their methods called only from the Swing event thread.
- To add an event to the event thread, create an object that describes a task to be performed in the event thread at some future time.
  - The object must be of type Runnable, so it can be executed in a thread.
- Then pass that object to the event thread using a synchronized method,
  `SwingUtilities.invokeLater()`
  that places it in the event thread's queue.
- Swing will execute the task when it can

Using `invokeLater()`

- How do we create an object on the event queue?

```
Runnable update = new Runnable() { // Anonymous
  public void run() {
    component.doSomething();  // Task
};
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.

Exercise 3: `JEventViewer`

- Read `JEventViewer`. It is almost identical to PressureController, but instead of writing the Phidget events to `System.out`, it adds them to a Swing `JTextArea`.
- Make the Phidget event thread modify a Swing component.

```
private JTextArea text;
  …
private void updateSensor(SensorChangeEvent se)
  { append(se.toString() + "\n" );
}
private void append( final String s ) {
  // Your code here:
  // Create an object of type Runnable
  //   Write its run() method to text.append( s );
  // Call invokeLater()
}
// String s must be final so Java knows it
// cannot change after the Runnable is created
```

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)`
- Otherwise use `InvokeLater()`

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012