

Introduction to Computation and Problem Solving

Class 34: Introduction to Threads

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

What is a Thread?

- Imagine a Java program that is reading large files over the Internet from several different servers. Some of these servers might be under heavy load or on slow connections to the Internet. Others might return the data quickly.
- Most of the time in our program will be spent waiting for input over the network. One programming approach is straightforward:

```
read file 1 from server A  
read file 2 from server B  
read file 3 from server C
```

....

What is a Thread?, 2

- Doing these reads sequentially is inefficient since the loading of file 2 from server B wouldn't start until all of file 1 is loaded.
- A much faster approach is to start reading from each file concurrently, and handle the partial files as they arrive.
- This requires the ability to have several tasks proceeding in parallel (as though they were each assigned to a separate independent processor).

3

What is a Thread?, 3

- Most computers still only have a single processor, so what we really want is an easy way for the program to switch among arriving data sources.
- More generally, we would like to be able to write programs where the "flow of control" branches, and the branches proceed in parallel.
- The processor can achieve this by switching between the different branches of the program in small time increments.
- This is the strategy behind *threads*.

4

Threads vs. Processes

- Most operating systems allow concurrent *processes* to proceed in parallel.
- *Processes* are expensive but safe. Processes are so well insulated from each other that it is both complex and often expensive to communicate between them.
- *Threads* are cheap, but different threads running in the same process are not well-insulated from each other.

5

Java Support for Threads

- Java is one of the few programming languages where the support for threads is a part of the language.
- Ada, a language developed by the Department of Defense, also has built-in support for threads, but Ada is little used outside DoD contexts.
- C# provides support similar to Java. In other languages such as C and C++, there are libraries to implement threads which are more or less standardized.

6

Java is Inherently Multithreaded

- In Java, the garbage collection of unreferenced objects is performed by the Java runtime system in a separate thread.
- Java also uses a separate thread to deliver user interface events. This allows a program to remain responsive even while it is involved in a long running calculation or I/O operation.
- Think how you would implement a "Cancel" function if you could not use threads.
- This means Java is inherently multithreaded. The Java runtime environment uses multiple threads even if the user's program doesn't.
- But programmers can also use threads in their own code. Our multiple file download strategy requires threads.

7

Keep It Simple

- The `Thread` class provides Java's support for threads.
- Less is always more with threads.
- You should always make your use of threads as simple as possible (but no simpler).

8

How Do I Tell a Thread What to Do?

There are two approaches:

1. You can subclass the Thread class and override the method `public void run()`.

```
public class MyThread extends Thread {
    public void run() {
        // code executed in the Thread goes here
    }
}
```

You can create an instance of this thread like this:

```
Thread t = new MyThread();
```

9

How Do I Tell a Thread What to Do?, 2

2. You can write a separate class that implements the Runnable interface, which contains only a single method:

```
public interface Runnable {
    public void run();
}
```

You create the Thread by using the Runnable as a constructor argument.

One reason to use this approach is that Java classes can only inherit from a single class. If you want to define a thread's `run()` method in a class that already inherits from another class, you can not use the first strategy.

10

Runnable Example

For example, consider the class `FrameInThread` defined as

```
public class FrameInThread
    extends JFrame implements Runnable {
    // constructors and other methods go here
    public void run() {
        // code executed in the Thread goes here
    }
}
```

If we wanted an instance of the `FrameInThread` class to run in its own `Thread`, we could use the statement

```
Thread t = new Thread(new FrameInThread() );
```

11

Starting and Stopping Threads

- **How do you start a thread working:** Call `start()` on the thread instance.

```
Thread t = new MyThread();
t.start();
```

- **How do you stop a thread and destroy it:** Let the `run()` method complete and the thread reference go out of scope or set the reference to null. The garbage collector will reclaim the thread's storage.
- `t.stop()` is "deprecated".

12

How to Tell If a Thread is Still Running

- You can ask it:

```
Thread t = new MyThread();
t.start();
. . .
if ( t.isAlive() ) // it's still running
else              // it isn't
```

- Or you can wait for it:

```
t.join(); // blocks until t completes
```

13

Simple Thread Example

- In this example we will implement the multithreaded download strategy discussed earlier.
- The program uses a separate `Thread` to read each URL from a Web server on the Internet and copy the contents of that URL to a local file.
- We call the class that does the work and extends the `Thread` class, `URLCopyThread`.
- `URLCopyThreadMain` creates a new instance of `URLCopyThread` for each copy operation.

14

URLCopyThreadMain

```
public class URLCopyThreadMain {

    public static void main(String argv[] ) {
        String[][] fileList = {
            {"http://www.nps.gov/brca/home.htm",
             "Bryce.pdf"},
            {"http://microscopy.fsu.edu/micro/gallery/dinosaur/dino1.jpg",
             "dino1.jpg"},
            {"http://www.boston.com/", "globe.html"},
            {"http://java.sun.com/docs/books/tutorial/index.html",
             "tutorial.index.html"},
        };
    };
}
```

15

URLCopyThreadMain, 2

```
for (int i=0; i<fileList.length; i++) {
    Thread th;
    String threadName = new String( "T" + i );
    th = new URLCopyThread( threadName,
                            fileList[i][0],
                            fileList[i][1] );

    th.start();
    System.err.println("Thread " + th.getName() +
                       " to copy from " + fileList[i][0] + " to " +
                       fileList[i][1] + " started" );
}
}
```

16

URLCopyThread

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

public class URLCopyThread extends Thread {
    private URL fromURL;
    private BufferedInputStream input;
    private BufferedOutputStream output;
    private String from, to;
```

17

URLCopyThread, 2

```
public URLCopyThread(String n, String f,
                     String t) {
    super( n );
    from = f; to = t;
    try {
        fromURL = new URL(from);
        input = new BufferedInputStream(
            fromURL.openStream());
        output = new BufferedOutputStream(
            new FileOutputStream(to));
    }
```

18

URLCopyThread, 3

```
catch(MalformedURLException m) {
    System.err.println(
        "MalformedURLException creating URL "
        + from);
}
catch(IOException io) {
    System.err.println("IOException " +
        io.toString() );
}
}
```

19

URLCopyThread, 4

```
public void run() {
    byte [] buf = new byte[ 512 ];
    int nread;
    try {
        while((nread=input.read(buf,0,512)) > 0) {
            output.write(buf, 0, nread);
            System.err.println( getName() + ": " +
                nread + " bytes" );
        }
    }
}
```

20

URLCopyThread, 5

```
input.close();
output.close();
System.err.println("Thread " + getName() +
    " copying " + from + " to " + to +
    "finished");
}
catch(IOException ioe) {
    System.err.println("IOException:" +
        ioe.toString());
}
} // end of run() method
} // end of URLCopyThread class
```

21

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*.

22

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 a `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 list in an inconsistent state.

23

synchronized Methods

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

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

means that `foo()` 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.

24

synchronized Method Cautions

- But note that `synchronized` methods only wait for other `synchronized` methods.
- Normal, unsynchronized methods invoked on the same object will proceed.
- And another thread can run another `synchronized` method on another instance of the same class.

25

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 (an instance of class `Class`).

26

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.
- As an example, the method of the Java AWT `Component` class that adds a `MouseListener` object to a `Component` (so that `MouseEvent`s are reported to the `MouseListener`) is 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)
```
- The classes in the Java Collections Framework do not, in general, use synchronized methods. There are methods in the `Collections` class, however, that produce synchronized views of collections, e.g.,

```
public static List synchronizedList(List list)
```

27

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.

28

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.

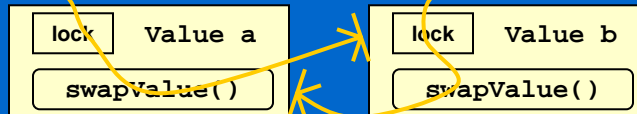
29

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);
    }
}
```

30

Deadlock Diagram



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

31

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.

32

Common Sense Rules for Threads

Some rules that may serve you well:

1. Only use multiple `Threads` when they are essential or when there are clear benefits to doing so.
2. Whenever you use multiple `Threads`, think carefully about whether the methods you wrote may need to be synchronized.
3. When in doubt, declare methods as `synchronized`.
4. If different runs of the same program with more than one thread execute very differently even though they are given the same inputs, suspect a synchronization problem.
5. If you use multiple `Threads`, try to make sure they die off as soon as they aren't needed.

33

Stopwatch Example

- As a more complicated example that illustrates an elegant interaction with `Swing` consider the `Clock` class that implements a stopwatch.
- `Clock` implements `Runnable` and so can be used to create its own `Thread`.
- `Time` is an inner class that supplies the stopwatch time display.

34

Clock, main()

```
public class Clock
  extends JFrame implements Runnable {
    private Thread clockThread = null;
    private Time time;
    private long accumTime = 0L;
    private long startTime = -1L;

    public static void main( String[] args ) {
        Clock clock = new Clock();
        clock.show();
    }
}
```

35

Clock, constructor

```
public Clock() {
    super( "Clock" );
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    JPanel buttons = new JPanel();
    JButton bStart = new JButton( "start" );
    bStart.addActionListener( new ActionListener()
        { public void actionPerformed( ActionEvent e )
          { start(); }
        } );
    // create buttons bStop and bReset same way
}
```

36

Clock, constructor, 2

```
buttons.add( bStart );
buttons.add( bStop );
buttons.add( bReset );
Container content = getContentPane();
content.add( buttons, BorderLayout.NORTH );
time = new Time();
content.add( time, BorderLayout.CENTER );
setSize( 240, 120 );
}
```

37

Clock, start()

```
private void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        startTime = System.currentTimeMillis();
        clockThread.start();
    }
}
```

38

Clock, stop(), reset()

```
private void stop() {
    if ( clockThread != null ) {
        clockThread = null;
        accumTime +=
            System.currentTimeMillis() - startTime;
    }
    time.repaint();
}

private void reset() {
    accumTime = 0L;
    startTime = System.currentTimeMillis();
    time.repaint();
}
```

39

Clock, run()

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        time.repaint();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e){}
    }
}
```

40

Clock, inner class Time

```
private class Time extends JPanel
{
    Font timeFont = new Font( "SansSerif",
                               Font.BOLD, 32 );
    private static final int timeX = 60;
    private static final int timeY = 40;

    public void paintComponent(Graphics g) {
        super.paintComponent( g );
        long ticks;
```

41

Time, paintComponent()

```
if ( clockThread == null )
    ticks = accumTime;
else
    ticks = System.currentTimeMillis() - startTime
            + accumTime;
long tenths = ticks/100L;
long seconds = tenths/10L; tenths %= 10;
long minutes = seconds/60L; seconds %= 60;
StringBuffer sb = new StringBuffer();
if ( minutes < 10 )
    sb.append( '0' );
sb.append( minutes );
```

42

Time, paintComponent(), 2

```
sb.append( ':' );
if ( seconds < 10 )
    sb.append( '0' );
sb.append( seconds );
sb.append( '.' );
sb.append( tenths );
g.setFont( timeFont );
g.drawString( sb.toString(), timeX, timeY );
} // end paintComponent
} // end Time
} // end Clock
```

43

Threads and Swing

All Java programs with a GUI 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.

44

JFileViewer

- If your program creates a GUI, but then just reacts to user input events, then you in effect trade one thread for another.
- But if you create a GUI and update it from your main thread, then you have to be more careful.
- Let's take a simple example. In Lecture 30, where we discussed streams, we discussed an example program called JFileViewer that reads in a text files and displays it in a class called JTextViewer.
- The way we did this was to read the entire text file, and then make the resulting text the initial contents of the JTextViewer.

45

JFileViewer Revisited

- A more interesting approach would have been to put up the interface and then to start reading the file, appending the new text as we read it off disk.
- The problem with this approach is that it involves modifying (calling methods on) objects in the GUI from a different thread than the event thread.

46

Threads and the AWT

- The initial Java GUI package, the AWT, synchronized many methods in the GUI classes to allow exactly this style of programming. 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.

47

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."

48

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.

49

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 callback 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.

50

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.

51

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.

52

Synchronized Swing Methods

As we have already mentioned and as you may have noticed in the Clock example there are some Swing methods that may safely be called from another thread. These include:

- `public void repaint()`
- `public void revalidate()`
- `public void addEventListener(Listener l)`
- `public void removeEventListener(Listener l)`

53

JBetterFileViewer

```
public void load( String path )
throws IOException {
    FileReader in = new FileReader( path );
    int nread;
    char [] buf = new char[ 512 ];

    while( ( nread = in.read( buf ) ) >= 0 ) {
        Update update = new Update( buf, nread );
        SwingUtilities.invokeLater( update );
    }

    in.close();
}
```

54

JBetterFileViewer, 2

```
private class Update
  implements Runnable {
  private final String theString;

  public Update( char [] b, int n ) {
    theString = new String( b, 0, n );
  }

  public void run() {
    theViewer.append( theString );
  }
}
```

55