Concurrency

Multiprocessing
Threads/Lightweight objects
Thread methods
Concurrency issues

What is program "execution"?

- In order to run, a program must be loaded into memory and given an initial state.
  - Code space, static variables, heap, stack (with main pushed on)
  - This is a process, job, or task.
- The Operating System allows multiple programs/processes to run simultaneously! How??
  - multi-tasking (multi-processing)
- How can we take advantage of this in our applications?
  - Consider Event-driven programming, for starters…
- Most corporations have several departments (say management, accounting, manufacturing, and sales).
  - For efficiency, each of these operations should be able to work on their tasks on their own, interacting only as necessary
  - Multitasking with multiple programs communicating with each other

What is concurrency?

- *Concurrency*: a property of systems in which several computational processes are executing at the same time, and potentially interacting with each other
  - In reality at the same time, may simply mean in any arbitrary order
- A program is executed as exactly one process per execution
- What if the people in accounting are really doing more than one (related) thing at a time?
  - copying the book, calculating the books, paying bills on the books, …
  - *Multi-threading*
- A program may find it efficient to then break up its job into one or more threads of execution (a.k.a. light-weight processes)
  - Share the same memory space
  - Each have their own execution stack
- Concurrency is at the heart of contemporary programming practices
Making Threads in Java

- Threads are built into Java (in java.lang.Thread) – no imports necessary!
- There are two ways to make classes run in separate threads in Java
  - (1) Extend the Thread class (includes abstract method: run())
    - Provides direct access to thread methods, but no other inheritance.
  - (2) Implement the Runnable interface (defines interface method: run())
    - Most common mechanism

A new thread can only be created by someone/somewhere calling
- (1) new subThread();        // subThread is a subclass of Thread
- (2) new Thread(myRunnable); // myRunnable's class implements Runnable

Example: Thread Race

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

public class ThreadRace {
    private int numberOfRacers;
    private static String winner = null;

    public ThreadRace (int numRacers) {
        this.numberOfRacers = numRacers;
    } // end constructor

    private void pause (int millisec) {
        try {
            Thread.sleep (millisec);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } // end method pause

    public static void setWinner(String winnerName) {
        if (winner == null) {
            winner = winnerName;
        }
    } // end method setWinner

    public void go () {
        JFrame frame = new JFrame("Thread Racer");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize (500,500);
        frame.setVisible (true);
        frame.setLayout (new GridLayout(numberOfRacers,1));
        int numberOfNonRacerThreads = Thread.activeCount();
        for (int i = 0; i < numberOfRacers; i++) {
            Racer racer = new Racer("Racer " + i);
            frame.add(racer);
            Thread thread = new Thread(racer);
            thread.start();
            while(Thread.activeCount() > numberOfNonRacerThreads) {
                pause(10);
            }
            System.out.println("The Race is over! " + winner + " wins!");
        }
    } // end method go

    public static void main (String[] args) {
        ThreadRace theRace = new ThreadRace(3);
        theRace.go();
    } // end method main
} // end class main
```
Thread Race

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

public class Racer extends JPanel implements Runnable {
    private String myName;
    private int myPosition = 0;
    private final int numberofSteps = 600;
    private final int racerHeight = 20;
    private final int racerWidth = 10;
    private int speed = 1;

    public Racer(String name) {
        myName = new String(name);
    }

    private void pause(int millisec) {
        //for (int i = 0; i < 1000000*millisec; i++) {
        // busy wait - do nothing
        //}
        try {
            Thread.sleep(millisec);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void run() {
        while (myPosition + racerWidth < numberofSteps) {
            myPosition += speed;
            repaint();
            pause(5);
        }
        ThreadRace.setWinner(myName);
        System.out.println("Racer " + myName + " has finished the race");
    }

    public void paintComponent(Graphics g) {
        g.setColor(Color.BLACK);       // set background
        g.fillRect(0,0,getWidth(),getHeight());
        g.setColor(Color.BLUE);        // draw line
        g.drawLine(0,getHeight()/2,
                   getWidth(),getHeight()/2);
        g.setColor(Color.RED);         // draw racer
        g.fillRect((myPosition*getWidth()/numberofSteps),
                   getHeight() - racerHeight/2,
                   racerWidth, racerHeight);
    }
}
```

**Thread swapping**

- How does a computer perform threading?
  - It can do multiple things at one time (if multi-processor)
  - It can appear to do multiple things at one time
- The computer has a clock that counts down a quantum for each executable process (and thread).
  - When process’s quantum expires, it is taking off the CPU
  - Its state is saved for later…
  - The scheduler decides which process gets the next quantum of time
  - How is this done fairly?
- The programmer can have impact on thread processing by setting priority
  - `int getPriority()` and `void setPriority(int)`
- In Java, the thread with the highest priority *always* gets the processor when it is not blocked

**Possible thread states**

- **New**
  - A thread instance has been created but not started
  - There is a Thread object but no thread of execution
- **Runnable**
  - eligible to run on a processor
  - has its own call stack and state
  - waiting to be assigned to a processor by the scheduler
- **Running**
  - actually on a processor
  - has been scheduled
  - will be removed from the processor and demoted to runnable after its quantum expires
- **Blocked**
  - Sleeping
  - Waiting
**Execution with threads of unequal priority**

```java
public void go () {
    JFrame frame = new JFrame("Thread Racer");
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    frame.setSize (500,500);
    frame.setVisible (true);
    frame.setLayout (new GridLayout(numberOfRacers,1));

    int numberOfNonRacerThreads = Thread.activeCount();
    for (int i = 0; i < numberOfRacers; i++) {
        Racer racer = new Racer("Racer " + i);
        frame.add(racer);
        Thread thread = new Thread(racer);
        thread.setPriority(thread.getPriority () + (2*i)-4);
        System.out.println("Racer " + i + " now has priority " + thread.getPriority());
        thread.start ();
    }

    while (Thread.activeCount() > numberOfNonRacerThreads) {
    }

    System.out.println("The Race is over! " + winner + " wins!");
}
```
Concurrency Issues

Synchronization

- What we need is a mechanism to identify operations which should NOT be interrupted by another thread.
- Such operations/methods are called atomic.
  - That is, they cannot be "divided" and must be done to completion before anyone can start the method.
- In Java, you can designate a method as being atomic by using the keyword `synchronized` in the method header.
  - How does Mutual Exclusion (Mutex) work?
    - Java provides a "lock" on each object.
    - An object becomes locked when a thread calls any of its synchronized methods. That thread gets the "key".
    - It requires a key to access any synchronized methods of a locked object.
    - The key is returned (and the object unlocked) when the thread execution leaves the synchronized method.

Concurrency Issues

- Unequal Priority (or unfortunate scheduling) can lead to starvation.
- Synchronization limits concurrency and can even lead to deadlock.

- The Dining Philosophers Problem
  - Dijkstra, 1971 (computational problem); Hoare (analogy)
  - Five poor (and hungry) philosophers are sitting around a table with a bowl of rice in front of each of them. There aren’t enough chopsticks to go around. One chopstick lies on the table between each philosopher, but two chopsticks are necessary to eat.
  - At most, how many can eat at a time?
  - What are the other philosophers doing?
  - What happens if a philosopher isn’t willing to give up his chopsticks until he is done eating?
  - What if each philosopher grabs exactly one chopstick?