Multiple Threading

G52APR Application programming

http://www.cs.nott.ac.uk/~jwl/G52APR.html

Michael Li – based on material from Colin Higgins and the Sun tutorial

Aims

- To understand the concept of concurrent execution, or concurrency
- To understand the process and thread
- To be able to create, execute, and terminate Threads
- To understand their States and their Priorities

Concurrency

- Multiple tasks for computer
  - Draw & display images on screen
  - Check keyboard & mouse input
  - Send & receive data on network
  - Read & write files to disk
  - Perform useful computation (editor, browser, game)
- How does computer do everything at once?
  - Multiple processing
  - Multitasking

Multiple processing

- On a multi-core machine
  - True concurrency
  - Multiple processes (threads)
  - Shared memory
- Generally much more complex than “normal” programming

Multitasking

- Approach
  - Time slicing
  - Computer does some work on a task
  - Computer then quickly switch to next task
  - Tasks managed by operating system (scheduler)
- Context switch

Concurrency

- On a single processor machine
  - Multitasking
  [Can improve performance by reducing waiting]
- On a multiple processors machine
  - Multiple processing
  - Multitasking
Performs multi tasks using

Process
- Definition – executable program loaded in memory
- Has own address space
  - Variables & data structures (heap and stack)
- Each process may execute a different program
- Communicate via operating system, files, network
- May contain multiple threads

Thread
- Definition – sequentially executed stream of instructions
- Shares address space with other threads
- Has own execution context
  - Program counter, call stack (local variables)
- Communicate via shared access to data
- Multiple threads in process execute same program
- Also known as “lightweight process”

Process vs. thread
- Processes
  - Program counter
  - Heap
  - stack
- Threads (Lightweight Processes)
  - Program counter
  - stack
- The JVM

Motivation for Multithreading
- Captures logical structure of problem
  - May have concurrent interacting components
  - Can handle each component using separate thread
  - Simplifies programming for problem
- Example

Web Server uses threads to handle ...

Multiple simultaneous web browser requests

Java Threads

- An interface: Runnable
  - Run()

- A class: Thread
  - Start()
  - Stop()
  - Sleep()
  - ... ...

Interface - Simple example

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        Thread tt = new HelloRunnable();
        tt.start();
    }
}
```
Multiple Threading

- Step 1: create your class (the thread)
- Step 2: implement the run() method
- Step 3: instantiate your class (a Thread type variable)
- Step 4: start the thread by calling start().

1. Creating a Thread

- Two ways to create a Thread:
  - 1. Implement Runnable (interface)
     - Implement the run() method.
  - 2. Extend Thread (sub-class)
     - Override the run() method.

- Due to single inheritance in Java, extending Thread means you cannot extend another class. The first alternative is more used.

Runnable

- Runnable is a very simple interface, it has just one method.
- The Thread's start() method calls the Runnable's run() method.

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

Thread constructors and names

- There are four basic constructors; their arguments can be:
  - A name (String) for the Thread (default null).
  - Should be unique, used for debugging etc.
  - An object (Runnable) to call a run() method in.
  - Default is this object

```
class ThreadExample implements Runnable {
    public ThreadExample {
        Thread t1 = new Thread();
        Thread t2 = new Thread("sun");
        Thread t3 = new Thread(this);
        Thread t4 = new Thread(this, "t4");
        String n2 = t2.getName();
        t3.setName("t3");
    }
    public void run() { }
}
```

Sub-classing (extending) example

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

2. Implementing Run()

```
class MyThread implements Runnable {
    MyThread() {
    }
    public void run() {
        // Does nothing?
        // Loop forever...
    }
    public static void main(String[] args) {
        Thread mythread = new Thread(new MyThread());
        mythread.start();
        myThread.stop();
    }
}
```
3. Executing Threads

- **run()**
  - Run it! (Runnable)
- **start()**
  - Activates the thread and calls run().
- **stop()**
  - Forcibly stops the thread.
- **suspend()**
  - Temporarily holds the thread.
- **resume()**
  - Resumes a halted thread.
- **destroy()**
  - Equivalent to UNIX’s “kill -9”
- **isAlive()**
  - Is it running?
- **yield()**
  - Let another thread run.
- **join()**
  - Waits for the death of a thread.
- **sleep(long)**
  - Sleep for a number of milliseconds.
- **interrupt()**
  - Interrupt sleep, wake up!

Sleep Example

```java
public class SleepMessages {
    public static void main(String[] args) throws InterruptedException {
        String info[] = {
            "Info one",
            "Info two"
        };
        for (int i = 0; i < info.length; i++)
            Thread.sleep(4000);
        System.out.println(info[i]);
    }
}
```

Sleep Example

```java
public class SleepMessages {
    public static void main(String[] args) throws InterruptedException {
        String info[] = {
            "Info one",
            "Info two"
        };
        for (int i = 0; i < info.length; i++)
            Thread.sleep(4000);
        System.out.println(info[i]);
    }
}
```

Threads race

```java
public class ThreadExample implements Runnable {
    public void run() {
        for (int i = 0; i < 3; i++)
            System.out.println(i);
    }
    public static void main(String[] args) {
        new Thread(new ThreadExample()).start();
        new Thread(new ThreadExample()).start();
        System.out.println("Done");
    }
}
```

Java Thread Example – Output

- Possible outputs
  - 0,1,2,0,1,2,Done // thread 1, thread 2, main()
  - 0,1,2,0,1,2,Done // thread 1, main(), thread 2
  - Done,0,1,2,0,1,2 // main(), thread 1, thread 2
  - Done,0,1,2,0,1,2 // main() & threads interleaved

Daemon Thread

- Why doesn’t the program quit as soon as Done is printed?
- Java threads types
  - User
  - Daemon
    - Provide general services
    - Typically never terminate
    - Call setDaemon() before start()
- Program termination
  - If all non-daemon threads terminate, JVM shuts down

JVM

- When running a JAVA program, how many processes are activated in memory?
  - Only one, JVM!!!
- When running a JAVA program, how many threads are active in memory?
  - It depends on JVM implementation and the functions of JVM used by the customer’s program.
  - The customer’s program can create threads
InterThread Communication

- Inter-thread communication methods are declared in `java.lang.Object`.
- Each object could be associated with a monitor (a sort of thread lock) – details in concurrency module.
- `wait()`
  - Suspend the thread.
  - Wait can also be time limited.
- `notify()`
  - Unlock the first monitored thread.
  - (The first that called `wait()` within the monitor.)
- `notifyAll()`
  - Unlocks all monitored threads.
  - Highest prioritised first!

Thread states

The suspended state

- A running Thread becomes suspended when:
  - It calls `sleep()` to tell the scheduler that it no longer wants to run;
  - It blocks for I/O; or
  - It blocks on entry to a synchronized object/method or in `wait()` for condition synchronisation.
  - It blocks waiting to `join()` another thread (when that thread finishes)

Volatile

- Consider the following code in thread's `run()`
  ```java
  Run()
  // currentValue is declared in main()
  currentValue = 5;
  for (;;) {
    System.out.print( currentVal + " ");
    Thread.sleep(1000); // System.out.print( currentVal + ",");
  }
  ```
- Another thread concurrently increases the value of `currentValue`
- Surprisingly, the thread will print 5,5,5,5,...

Volatile

- Since the value of `currentValue` is not modified in the loop, the compiler simply continues printing the same value
- Define the variable as `volatile` by:
  ```java
  volatile int currentValue;
  ```
- This will cause to perform actual memory access and value update for every access to the variable
Thread priorities

- Threads have priorities which heuristically influence schedulers:
  - each thread has a priority in the range Thread.MIN_PRIORITY to Thread.MAX_PRIORITY
  - by default, each new thread has the same priority as the thread that created it—the initial thread associated with a main method by default has priority Thread.NORM_PRIORITY
  - the current priority of a thread can be accessed by the method getPriority and set via the method setPriority.
- When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities.

Summary

- Concurrency
- Process and thread
- Creation of Threads
- Thread Constructors and names
- Execution of Threads
- Thread States
- Priorities

Any questions?

- No lecture on Thursday!!!