Java Threads
Written by John Bell for CS 342, Spring 2018

Based on chapter 9 of “Learning Java, Fourth Edition” by Niemeyer and Leuck, and other sources.

Processes

• A **process** is an instance of a running program.
  • **Programs** are stored on the hard drive.
  • **Processes** run in memory.
    • If the same program is running more than one time, each instance is a separate process.

• Each process has a unique Process ID (PID), owner, blocks of allocated memory, and allocated process resources.
  • (It is possible, under certain circumstances, for processes to share memory and other resources, such as read-only blocks of executable code.)

• The **address space** of each process is unique and private, regardless of what it may share with others.
Threads

- Threads are separate flows of execution, running within a single process, and sharing a common address space.
- Threads share a common PID and system resources, such as the open file table.
- Typical tasks assigned to separate threads include background file saving or spell checking, animations, event handling, keyboard processing, graphics updating, etc. Network communications are also commonly threaded.

Shared Thread Access (to private data.)

- Shared access between threads is determined by scope, as well as privacy.
  - Two threads running code in different instances of the same class will have shared access to class static data, but not instance data.
  - Two threads running code in the same instance of a class will have access to shared instance as well as class data.
  - Each running method has its own stack space containing separate local variables. (Similar to recursion.)
  - Threads running code not within the same class will only have access to shared global data and resources, (in languages that support global data.)
Time Slicing

• Multiple processes appear to run concurrently through time-slicing: Rapidly switching from one process to another in very short time slices.

• Threads run concurrently through sharing and slicing at several levels:
  • If multiple CPUs are present, each runs independently, and the OS schedules kernel threads on each CPU.
  • The OS time-slices multiple kernel threads per CPU.
  • The OS time-slices multiple user threads on each kernel thread.

• Thread scheduling is unknowable and largely uncontrollable from user code.

Implementing Threads in Java

• Implementing Threads in Java requires 2 things:
  • An instance of the Thread class ( or a descendant. )
  • An instance that implements the Runnable interface.
The Runnable Interface

• Defines a single method: `void run();`
  • The run method may not take any parameters, return any values, or throw any unchecked exceptions.

• Calling the `start()` method of a Thread causes the system to call the `run()` method of the Runnable.

• (Threads in Java are not automatically started upon creation, as opposed to C++ pThreads.)

Java Thread Class

• Constructors may take no arguments, or some combination of:
  • An instance of Runnable,
  • A String name,
  • A ThreadGroup

• The Thread class implements Runnable, with a do-nothing `run()` method, which can be overridden in descendant classes.
  • If a Thread is constructed **without** a separate Runnable, then it acts as its own Runnable.
  • If a Thread is constructed **with** a separate Runnable, then calling `run()` on the Thread calls `run()` on the separate Runnable.
Sample I - Separate Classes

- **class Animation** implements Runnable {
  
  boolean animate = true;
  
  public void run() {
    
    while (animate) {
      // draw Frames
      
      ... 
    }
  }
}

- Animation happy = new Animation("Mr. Happy");
  Thread myThread = new Thread( happy );
  myThread.start();

Sample II - Runnable creates and starts its own thread

- **class Animation** implements Runnable {
  
  Thread myThread;
  
  Animation (String name) {
    
    myThread = new Thread( this );
    myThread.start();
  }
  
  ... // run( ) method down here somewhere
}

- Alternate: Thread could be created and/or started in a separate method besides the constructor.
Sample III - Thread IS Runnable

• class Animation extends Thread {
  boolean animate = true;
  public void run() {
    while (animate) {
      // draw Frames
      ...
    }
  }
}

• Creating an Animation creates a Runnable Thread.

Controlling Thread Execution

• Scheduling of Threads is entirely at the whim of the OS. There is no real way to control how fast or slow different threads run, or to guarantee order of operations for activities occurring in different threads. However there is some means of control:
  • sleep( ) causes the thread to pause for a time.
  • wait( ) and join( ) are used to coordinate threads.
  • interrupt( ) wakes up a thread blocked by sleep( ), wait( ), or a blocking I/O operation.
  • Returning from run( ) is the best way to end a thread.
  • (Deprecated: stop( ), suspend( ), resume( ) )
Sleep Example

```java
try {
    // The current thread
    Thread.sleep(1000); // milliseconds, i.e. 1 sec
} catch (InterruptedException e) {
    // someone woke us up prematurely
}
```

Thread.join( )

- Causes the current thread to pause until a separate thread finishes.
- Thread background = new Thread ( myRunnable );
  background.start( );

  // Do stuff of my own for a while
  ...
  background.join( ); // Pause till background finishes.
  ...
  // Now use the results ( if any ) of background.
- join(int) puts a limit on how long to pause. (msec.)
Synchronizing access to data

• One of the big problems with concurrent code is synchronizing access to shared data, so that only one thread modifies it at a time.
• The standard approach is to lock a data item or block of code, so that only one thread at a time can access that data or run that code.
• Every Object in Java has an associated lock (monitor), that can be used for this purpose, either through the `synchronized` keyword or the `wait()` and `notify()` methods of Object.

The `synchronized` keyword

• A method in Java can be labelled with the keyword `synchronized`, guaranteeing that only one instance of the method will run at a time.
  • `synchronized` int sumRow() {
      return cellA1 + cellA2 + cellA3;
  }
  • If the method is static, a class-level lock is applied.
  • If the method is not static, an instance-level lock is applied.
  • Multiple synchronized methods in the same class share the same lock.
• Alternatively any block of code can be synchronized, by locking on any object:
  • `synchronized` (myObject) {
      // Functionality that needs exclusive access to resources
  }
wait( ), notify( ), and notifyAll( )

- There are times when synchronized code may want to voluntarily wait, and let other code have access until something occurs.
- `wait( )` releases a lock, and sleeps until either a timer expires or it is notified.
  - Upon reawakening, the code needs to reacquire the lock before it can proceed.
- `notify( )` will wake up one sleeping thread that had blocked on `wait( )`. However it does not give up the lock. `notifyAll( )` wakes up all such sleeping threads.

Thread State

- At any given time, a Thread can be in one of five defined states:
  - NEW - Created, but not yet started.
  - RUNNABLE - Normal running state, even when blocked on an I/O operation such as `read( )`.
  - BLOCKED - Blocked waiting to enter a section of synchronized code.
  - WAITING, TIMED_WAITING - waiting on another thread, via a call to `join( )` or `wait( )`
  - TERMINATED - Completed, due to returning, an unhandled exception, or a call to (deprecated)`stop( )`. 
Thread Priorities

• The `setPriority(int)` method allows programmers to set the relative priorities of different threads.
• These should be considered as suggestions to the system, not hard guarantees. Due to the complexities of the underlying operating system, there is no way to rigidly enforce or guarantee any kind of (relative) performance when using threads.

Thread Groups

• Threads may be assembled into groups (and subgroups) using `ThreadGroup` objects.
• The purpose of thread groups is to apply controls to entire groups of threads at one time.
• All of the methods available in `Thread()` are available in `ThreadGroup()`, and are applied to all the threads in the group.