Threads 101

Concurrent programming techniques have been around for many years, but have not seen widespread use until recently for a number of reasons. Traditional concurrent programs used multiple processes that required expensive inter-process communication (IPC) mechanisms and were tremendously platform dependent. If you could `fork` with the best of them on UNIX, that wouldn't get you anywhere on VAX/VMS. Furthermore, creating processes on UNIX was enough of a performance hit, but on many other operating systems it was abysmally slow and to be avoided whenever possible. Operating system designers got the message and invented lightweight processes, which evolved into what we know as threads today.

A thread is an execution path through a program, and there can be many threads active in a single process, all sharing the same process space. Using threads solved the performance hit of creating and managing separate processes every time concurrency was called for, but there remained the portability issue. The solution was of course to make threads part of the programming language, instead of leaving the job to the operating system or competing library vendors. This is what Java has achieved with a large degree of success.

Threaded programming is a complex affair. In this article I'll scratch enough of the surface to make you comfortable managing independent tasks and shared resources concurrently in Java.

The Benefits of Concurrency

The motivation for concurrent programming is simple: better throughput. For example, separating the tasks of computation and presentation into distinct threads makes for a more responsive interactive program, because users can still press buttons and fill in text fields while awaiting the result of a previous request. On systems with multiple processors separate threads can be scheduled to run simultaneously. Even on single processor systems, good thread scheduling algorithms make separate tasks appear to run concurrently to the human eye. A web server, for example, spawns a separate threads to process incoming HTTP requests so one request doesn't have to wait for an unrelated one to finish before it gets a response. A web browser can use separate threads to display different parts of a web page.

Programmers like threads because they "simplify" code. That may sound a bit contradictory in light of the first sentence two paragraphs above. A language that supports concurrency shifts the details of managing locks and communication paths from your code to your platform, so all you need to do is to define your tasks, along with a minimal amount infrastructure. Getting the glue between threads to work is the tricky part, but the code comes out clean.

The program in Listing 1 shows one way to create and launch separate threads, which is to extend the `Thread` class and override the `run` method. The `MyThread` class constructor expects a message to display and a count representing how many times to display it. The `Thread` class has a constructor that accepts an optional thread name. This feature isn't used much in practice, but I use it here to hold the message each thread is supposed to repeatedly print to the console screen. My `run` method just displays that message `count` times. To launch a

thread I call `Thread.start`, which in turn invokes `MyThread.run` automatically. As you can see, all but one of the repetitions of "DessertTopping" appear before the "FloorWax" thread gets a chance to execute. Your output may vary, and is a function of the thread-scheduling algorithm used on your Java platform.

Sometimes you may want a thread to explicitly give other threads a chance to run, instead of hogging all the processor time it possibly can. You can do such cooperative multitasking with the `Thread.yield` static method, as illustrated in Listing 2. The net effect here is that each thread just takes it turn displaying a single instance of its message.

Another way to let threads share processor time is to pause them for a period of time. The static method `Thread.sleep` suspends the thread for at least the number of milliseconds specified by its argument so other threads can run (see Listing 3). `Thread.sleep` can throw an `InterruptedException` (explained below), but it won't happen in this example, so I just ignore it. This time the interleaving of the output appears more random than in the previous examples.

How many threads do you suppose are active at any one time in the examples above? The answer is 1, 2, or 3 (not 0, 1, or 2). When you start the virtual machine with the command

```
java Independent
```

the `main` method executes in an initial thread, called the *main thread*. Listing 4 illustrates this by displaying information about each thread. After `t1` starts there are two threads active, and finally three after `t2` begins. To determine all the threads in a program requires access to their thread groups. All threads belong to a thread group[1]. Since I didn't explicitly create any thread groups, all threads in Listing 2 belong to the default thread group (named "main"). The method `ThreadGroup.activeCount` returns the total number of threads in the group that haven't yet terminated, and `ThreadGroup.enumerate` fills an array with the all the threads currently running and returns the count of the same. `Thread.join` waits until its thread completes, if necessary, before returning. The `ThreadGroup.toString` method displays a thread's name, priority[2], and thread group.

Another way to define a thread is to first define a task to implement the `Runnable` interface, which defines a single `run` method, and then to pass that task to an alternate `Thread` constructor, as shown in Listing 5. This approach is attractive for a couple of reasons. From a conceptual point of view, it makes sense to separate a task from the thread that runs it. From a more pragmatic perspective, you need this technique if your task class extends another class, since Java only supports single inheritance.

Thread States

You might be wondering how the output in the previous examples came out the way it did. Why didn't the characters of the individual lines get interleaved? The reason is that the methods in `java.lang.io` cause a thread to *block*, which means the thread is put on hold until the call is

complete. If you were to rewrite Listing 3 to print the characters individually using `print` instead of `println`, then such interleaving does occur (see Listing 6).

Being blocked is just one of the several states a thread can be in. As Figure 1 illustrates, when a thread is started, it enters the Runnable (or Ready) state, which means it is eligible to be chosen to execute by the JVM's thread scheduler. If the thread calls a blocking method, it enters the Blocked state until the blocking operation is complete, after which it is eligible to be rescheduled. Similarly, when a thread's code encounters a call to `Thread.sleep`, it enters the Sleeping state for the specified amount of time, and then becomes Runnable. When the scheduler suspends a thread so another can run, the first thread moves from the Running to the Runnable state. Only Runnable threads can be scheduled for execution. A thread is considered dead when its run method returns. Once dead, a thread cannot be restarted, although its associated `Thread` object is still available until it is garbage-collected.

The acceptable way to stop a thread is to politely ask it to stop itself. A typical situation where you need to stop a thread prematurely is in interactive applications. For example, if a database query is taking too long and you want to abort and do something else in your application, you need to be able to press a Cancel button and have the query stop. The program in Listing 7 contains a first attempt to stop a thread. The main thread spawns a worker thread that displays a count every second, and then waits for user to press Enter. The `Counter` class has a `cancel` method that sets the boolean field `cancelled`. The loop in `Counter.run` checks `cancelled` before printing on each iteration so it knows when to exit. I'll explain shortly why this is not a generally safe technique for stopping a thread.

Did you notice that the main thread terminated before the worker thread did in Listing 7? The application keeps running as long as any threads are still active. Well, almost any thread. There are two types of threads in Java: *user* threads and *daemon* (pronounced "DEE-mun") threads. The threads I've shown you so far are user threads. The difference between the two thread types is that if there are only daemon threads active, the application can terminate. You make a daemon thread by simply calling `setDaemon(true)` on an existing thread. By making the `Counter` thread in the previous example a daemon thread I can halt the application without the canceling mechanism (see Listing 8).

A very common technique for stopping a thread is to use `Thread.interrupt`, which is an official way of getting a thread's attention. When you interrupt a thread, it doesn't really stop – it just causes a call to `Thread.interrupted` to return `true` when invoked in that thread. The program in Listing 9 uses this feature to kill the `Counter` loop. This program clarifies why `Thread.sleep` may throw an `InterruptedException`, and what to do about it. If a thread is sleeping, it can't be directly interrupted, since it isn't running. Instead, it moves to the Runnable state and waits its turn to execute. When it runs, execution resumes in the `InterruptedException` handler associated with the pending call to `Thread.sleep`. The usual thing to do is to call `interrupt` again, since the initial call gave way to the exception. The loop will then complete on its next iteration and `run` will return. (Notice in this example that I reverted to extending `Counter` from `Thread`. If you use the `Runnable` approach, just call `Thread.getCurrentThread().interrupt()`).

Thread Safety

One of the most difficult concepts to grasp in concurrent programming is how to share resources among separate tasks. Whenever two or more threads access the same object, you need to ensure that those threads take turns; otherwise your data can end up in an inconsistent state. Code that guards against this phenomenon is said to be *thread safe*. To illustrate, consider a Book object in a program that supports a public library. Suppose each book has title, author, and borrower fields. If two threads try to check out the same book at the same time, there's no telling what could happen. Perhaps the thread that executed last would win the privilege of being recorded as the borrower, but both threads would think they completed the transaction successfully. Bad news.

The Java threading model uses *monitors* to prevent threads from accessing shared data simultaneously. A monitor is a mechanism that establishes what are known as *critical sections* in code. Only one thread is allowed in a critical section at a time. A monitor is always associated with an object. The monitor for an object protects all the critical sections in the code associated with that object as a unit (so one thread can't be in one critical section while another thread is in another section). You declare critical sections with the `synchronized` keyword. Synchronization is expensive (four to six times more than non-synchronized methods), so you should use it sparingly

The only thread-sensitive data in Listing 10 is the `borrower` field. The other two fields are read-only and are therefore inherently thread-safe. To protect access to `borrower`, I do two important things: 1) declare it to be `private` (otherwise there is no protection whatever!) and 2) declare all methods that use it `synchronized` – including the accessor method `getBorrower` (otherwise it would be possible for one thread to get the borrower right before another thread changes it, making the data invalid). To execute one of these methods, a thread must obtain a lock on the associated object. (All Java objects have a hidden lock field). If another thread has the lock, the requesting thread waits until it becomes available. Once a thread has the lock (which is called being "in the monitor"), no other thread can execute any synchronized code. The thread in the monitor has exclusive access to the synchronized block until it leaves it and releases the lock. This is why `checkout` and `checkIn` can call `isAvailable` without any problems.

Since it is always a good idea to keep critical sections as small as possible, you should declare an entire method synchronized only when absolutely necessary. The program in Listing 11 shows how to declare a synchronized block, while at the same time fixing the problem with stopping a thread via the `cancel` method in Listing 7. Since multiple threads executing Counter objects could conceivably be running at the same time, then the data fields need to be protected. Since cancel is a one-line method, there is no problem declaring it synchronized, but `run` must not be synchronized. Since it has an infinite loop, it would never be interrupted if it ran in the monitor. For this reason I place only the two statements that require protection in a synchronized block on the current object (remember, a monitor always requires an object to lock).

It is possible to declare code in static methods as a critical section. The associated monitor simply obtains a lock on the class object. Listing 12 illustrates this by fixing the interleaved output problem with Listing 7. There is no shared data in this case, so no non-static synchronized methods are necessary. Instead, I just make the static method display synchronized. When MyThread.run calls MyThread.display, the thread obtains a lock on the object MyThread.class, so the output proceeds uninterrupted. If I were to make display non-static it would not prevent the interleaving, because the locks would be on two unrelated objects (`t1` and `t2`).

Communication Between Threads

Synchronization keeps threads from getting in each other's way, but it doesn't allow threads to talk to each other. Sometimes a thread needs to wait on a condition brought about by another thread. Such is the case in typical producer-consumer scenarios, like real-time queuing systems.

Listing 1 – File Independent.java: Illustrates Independent Threads

```
class MyThread extends Thread
{
   private int count;
   public MyThread(String msg, int count)
   {
      super(msg); // Optional thread name
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
         System.out.println(getName());
   }
}

class Independent
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 8);
      Thread t2 = new MyThread("FloorWax", 4);
      t1.start();
      t2.start();
   }
}

/* Output:
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
DessertTopping
FloorWax
FloorWax
FloorWax
FloorWax
```

```
DessertTopping
*/
```

Listing 2 – File Independent2.java: Uses the `Thread.yield` method

```
class MyThread extends Thread
{
   private int count;
   public MyThread(String msg, int count)
   {
      super(msg); // Optional thread name
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         System.out.println(getName());
         Thread.yield();
      }
   }
}

class Independent2
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 8);
      Thread t2 = new MyThread("FloorWax", 4);
      t1.start();
      t2.start();
   }
}

/* Output:
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
DessertTopping
DessertTopping
DessertTopping
*/
```

Listing 3 – File Independent3.java: Uses the `Thread.sleep` method

```
class MyThread extends Thread
{
   private int delay;
   private int count;
   public MyThread(String msg, int delay, int count)
   {
      super(msg); // Optional thread name
```

```
      this.delay = delay;
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         try
         {
            Thread.sleep(delay);
            System.out.println(getName());
         }
         catch (InterruptedException x)
         {
            // Won't happen in this example
         }
      }
   }
}

class Independent3
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 100, 8);
      Thread t2 = new MyThread("FloorWax", 200, 4);
      t1.start();
      t2.start();
   }
}

/* Output:
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
*/
```

Listing 4 – File ListThreads.java: Illustrates the Default Thread Group and Thread.join

```
class ListThreads
{
   public static void list(String label)
   {
      ThreadGroup group = Thread.currentThread().getThreadGroup();
      int nActive = group.activeCount();
      System.out.println("\n" + label + ":\n=====");
      System.out.println("nActive = " + nActive);
      Thread threads[] = new Thread[nActive];
```

```
      int nThreads = group.enumerate(threads);
      System.out.println("nThreads = " + nThreads);
      for (int i = 0; i < nThreads; ++i)
         System.out.println(threads[i]);
      System.out.println();
   }

   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 200, 10);
      Thread t2 = new MyThread("FloorWax", 300, 7);
      list("Before");
      t1.start();
      t2.start();
      list("During");

      try
      {
         // Wait for worker threads to finish:
         t1.join();
         t2.join();
      }
      catch (InterruptedException x)
      {
         // won't happen
      }
      list("After");
   }
}

/* Output:

Before:
=====
nActive = 3
nThreads = 1
Thread[main,5,main]


During:
=====
nActive = 3
nThreads = 3
Thread[main,5,main]
Thread[DessertTopping,5,main]
Thread[FloorWax,5,main]

DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
```

```
DessertTopping
FloorWax
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax

After:
=====
nActive = 1
nThreads = 1
Thread[main,5,main]

*/
```

Listing 5 – File Independent4.java: Illustrates Runnable Objects

```
class MyTask implements Runnable
{
   private int delay;
   private int count;
   private String name;
   public MyTask(String name, int delay, int count)
   {
      this.delay = delay;
      this.count = count;
      this.name = name;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         try
         {
            Thread.sleep(delay);
            System.out.println(name);
         }
         catch (InterruptedException x)
         {
            // Won't happen in this example
         }
      }
   }
}

class Independent4
{
   public static void main(String[] args)
   {
      Thread t1 = new Thread(new MyTask("DessertTopping", 100, 8));
      Thread t2 = new Thread(new MyTask("FloorWax", 200, 4));
      t1.start();
      t2.start();
   }
}
```

Listing 6 – File Independent5.java: Reveals Interleaved Output

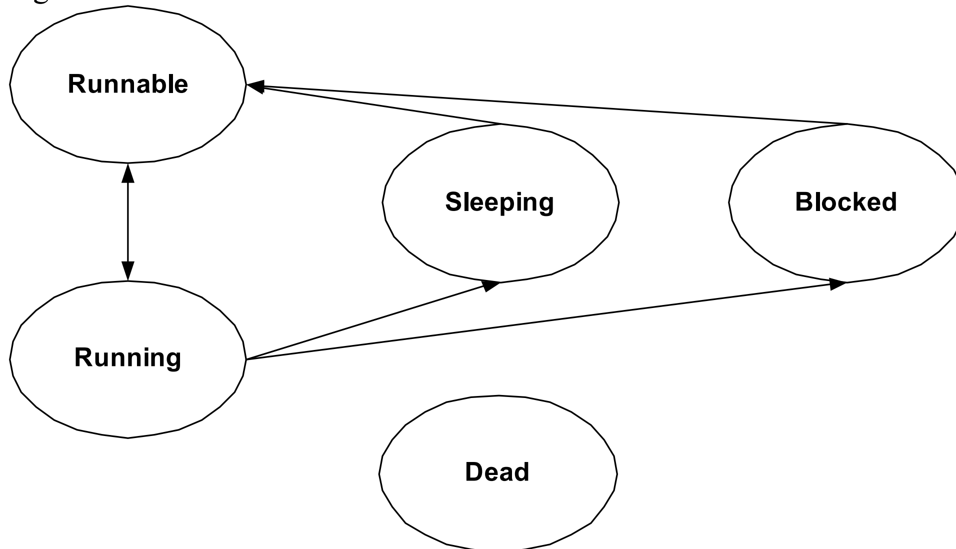
```
class MyThread extends Thread
{
   private int delay;
   private int count;
   public MyThread(String msg, int delay, int count)
   {
      super(msg); // Optional thread name
      this.delay = delay;
      this.count = count;
   }
   public void run()
   {
      for (int i = 0; i < count; ++i)
      {
         try
         {
            Thread.sleep(delay);
            display();
         }
         catch (InterruptedException x)
         {
            // Won't happen in this example
         }
      }
   }
   void display()
   {
      String s = getName();
      for (int i = 0; i < s.length(); ++i)
         System.out.print(s.charAt(i));
      System.out.println();
   }
}

class Independent5
{
   public static void main(String[] args)
   {
      Thread t1 = new MyThread("DessertTopping", 100, 8);
      Thread t2 = new MyThread("FloorWax", 200, 4);
      t1.start();
      t2.start();
   }
}

/* Output:
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorDessertTopping
Wax
DessertTopping
FloorWax
DessertTopping
```

```
DessertTopping
FloDessertTopping
orWax
*/
```

Figure 1 – Thread States



Listing 7 – File Cancel.java: Stops a Thread (but not safely – see Listing 11)

```
import java.io.*;

class Counter implements Runnable
{
   private int count = 0;
   private boolean cancelled = false;

   public void run()
   {
      while (!cancelled)
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x){}
      }
      System.out.println("Counter Finished");
   }
   void cancel()
   {
      cancelled = true;
   }
}
```

```
class Cancel
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      new Thread(c).start();

      try
      {
         System.in.read();
      }
      catch (IOException x)
      {
         System.out.println(x);
      }
      c.cancel();      // Don't forget this!
      System.out.println("Exiting main");
   }
}

/* Output:
Press Enter to Cancel:
0
1
2

Exiting main
Counter Finished
*/
```

Listing 8 – File Cancel2.java: Illustrates a Daemon Thread

```
// Illustrates a Daemon Thread

import java.io.*;

class Counter implements Runnable
{
   private int count = 0;

   public void run()
   {
      for (;;)
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x){}
      }
   }
}
```

```
class Cancel2
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      Thread t = new Thread(c);
      t.setDaemon(true);
      t.start();

      try
      {
         System.in.read();
      }
      catch (IOException x)
      {
         System.out.println(x);
      }
     System.out.println("Exiting main");
  }
}

/* Output:
Press Enter to Cancel:
0
1
2

Exiting main
*/
```

Listing 9 – Uses Thread.interrupt to kill a Thread

```
import java.io.*;

class Counter extends Thread
{
   private int count = 0;

   public void run()
   {
      while (!interrupted())
      {
         System.out.println(count++);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x)
         {
            interrupt();
         }
      }
      System.out.println("Counter Finished");
   }
}
```

```
class Interrupt
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      c.start();

      try
      {
         System.in.read();
      }
      catch (IOException x)
      {
         System.out.println(x);
      }
      c.interrupt();
      System.out.println("Exiting main");
   }
}

/* Output:
Press Enter to Cancel:
0
1
2

Exiting main
Counter Finished
*/
```

Listing 10 – File Book.java: Illustrates Synchronized Methods

```
class Book
{
   private final String title;
   private final String author;
   private String borrower;

   public Book(String title, String author)
   {
      this.title = title;
      this.author = author;
      borrower = null;
   }

   public synchronized boolean checkOut(String borrower)
   {
      if (isAvailable())
      {
         this.borrower = borrower;
         return true;
      }
      else
         return false;
   }
```

```
   public synchronized boolean checkIn()
   {
      if (!isAvailable())
      {
         borrower = null;
         return true;
      }
      else
         return false;
   }

   public String getTitle()
   {
      return title;
   }

   public String getAuthor()
   {
      return author;
   }

   public synchronized boolean isAvailable()
   {
      return borrower == null;
   }

   public synchronized String getBorrower()
   {
      return borrower;
   }
}
```

Listing 11 – File Cancel3.java: Illustrates a Synchronized Block

```
import java.io.*;

class Counter implements Runnable
{
   private int count = 0;
   private boolean cancelled = false;

   public void run()
   {
      for (;;) // Loop must not be synchronized!
      {
         synchronized(this)
         {
            System.out.println(count++);
            if (cancelled)
               break;
         }
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException x){}
      }
```

```
      System.out.println("Counter Finished");
   }
   public synchronized void cancel()
   {
      cancelled = true;
   }
}

class Cancel3
{
   public static void main(String[] args)
   {
      System.out.println("Press Enter to Cancel:");
      Counter c = new Counter();
      new Thread(c).start();

      try
      {
         System.in.read();
      }
      catch (IOException x)
      {
         System.out.println(x);
      }
      c.cancel();      // Don't forget this!
      System.out.println("Exiting main");
   }
}

/* Output:
Press Enter to Cancel:
0
1

Exiting main
2
Counter Finished
*/
```

Listing 12 – File StaticLock.java: Illustrates a Class-level Lock

```
class MyThread extends Thread
{
    private int delay;
    private int count;
    public MyThread(String msg, int delay, int count)
    {
        super(msg); // Optional thread name
        this.delay = delay;
        this.count = count;
    }
    public void run()
    {
        for (int i = 0; i < count; ++i)
        {
```

```
            try
            {
                Thread.sleep(delay);
                display(getName());
            }
            catch (InterruptedException x)
            {
                // Won't happen in this example
            }
        }
    }
    synchronized static void display(String s)
    {
        for (int i = 0; i < s.length(); ++i)
            System.out.print(s.charAt(i));
        System.out.println();
    }
}

class StaticLock
{
    public static void main(String[] args)
    {
        Thread t1 = new MyThread("DessertTopping", 100, 8);
        Thread t2 = new MyThread("FloorWax", 200, 4);
        t1.start();
        t2.start();
    }
}

/* Output:
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
DessertTopping
FloorWax
DessertTopping
*/
```

---

[1] ThreadGroups can contain other ThreadGroups, forming a tree with the default group as root.

[2] Priorities are intergers in the range [Thread.MIN_PRIORITY, Thread.MAX_PRIORITY]. What these priorities actually mean depends on your platform. Using *native threads*, where the JVM uses the underlying operating systems threading facilities for concurrency, the number of priorities may differ, so two priority numbers can be mapped to the same native priority. The only assumption you can really make is that threads with higher priority may be executed in preference to threads with lower priority.