CS2 Advanced Programming in Java note 9

Concurrent Programming in Java

This note and the next introduce concurrent programming in Java. After working through this one you should be able to say why concurrent programming is useful, and make simple use of Java threads.

Introduction

You probably already know how a multitasking operating system shares resources between several processes running on a single machine. Clearly this is vital for a multiuser system, but it offers other benefits too.

- Efficient use of mixed resources. While one program is waiting for a disk to spin, another can get on with some computation.
- Responsiveness. A program can begin processing a user request without having to wait for everything else to finish first. Similarly, the system can swiftly respond to a hardware interrupt.
- Flexibility. In a windowed GUI, for example, several applications can all be ready to receive input, with the user free to choose between them.

These aren't just for several applications sharing one processor; they also make sense within a single program too — think about an impatient user interacting with a web browser. Writing an application that uses several simultaneous strands of control is *concurrent programming*. Some programming languages provide explicit support for concurrency; many more do not, which can make life difficult.

Within a single application we usually refer to the strands of control as *threads* rather than processes. There are various differences in how threads and processes operate. Typically, processes are explicitly supported by the operating system, which controls switching between them; while threads might just be simulated by a programming language environment. Moreover, different processes use separate memory segments and communicate with one another through pipes, files, or other mechanisms provided by the operating system; while threads often share memory and can communicate directly by reading and writing this memory.

The degree to which threads are actually concurrent varies between platforms, depending on how aggressively control is transferred from one to another.

- Non-preemptive scheduling (*coroutines*): threads only lose control when they explicitly give it up.
- Preemptive scheduling: control may switch between threads at any time.
- Simultaneous execution: on a multiprocessor system there may truly be more than one thread running at the same time.

The code that oversees this is a *scheduler*; as well as deciding when to switch control, it will have some policy about which thread to run next.

Concurrent programming takes advantage of the ability to several things at once, even if that means sharing time slices. This is rather different from *parallel programming*, where the main goal is to speed up large computations by spreading them over several real processors. However, concurrent and parallel programmers do of course have some common concerns.

Java threads

Java supports concurrent programming as an integral part of the language. This means that threading is always available, though how concurrent this is can vary from one implementation to another: all the way from non-preemptive scheduling ("after you"; "no, after you") right up to simultaneous execution on multiprocessors. This "slackness" in the definition of Java is intended to allow implementers maximum freedom to take advantage of whatever features their platform offers. However, the downside of this is that a program which works in one way on one platform might work quite differently on another platform, unless the programmer takes great care to avoid this. (Bugs in concurrent programs are frequently very subtle and can be *very* hard to detect!!)

A running thread is an instance of class Thread, and code can always find out what thread it is in with the class method Thread.currentThread(). A fresh thread is created with **new** like any other object; it then needs to be started off running.

```
Thread t = new Thread();                      // Get a fresh thread
 t.start ();                                  // Set it going
 ...                                          // Do my own thing
 t.join ();                                   // Wait for thread t to finish
```

In the middle here there are two threads running concurrently: the original one, and an additional thread t. Sadly, t won't actually do any computation unless we assign it some code.

Calling the start method on any thread will in turn invoke its run() method. By default, this does nothing, but a subclass of Thread can override it to do something useful (another good example of dynamic binding in use).

```
class MessageThread extends Thread {
  private String message;
  public MessageThread (String m) { message = m; }
```

```
    public void run () {  System.out.println(message); }
  }
```

The following code prints two messages in an indeterminate order.

```
Thread mt = new MessageThread("That␣thread");
mt.start ();
System.out.println ("This␣thread");
```

A subclass of Thread can also override other methods to add behaviour — for example, by putting initialization code into start ().

The life-cycle of a thread

A Java thread has five states: born, running, runnable, dormant, and dead. (There are actually a few others, which we won't worry about here.)

- Once a thread is *born*, starting it makes the thread *runnable*.
- The scheduler switches the thread between *runnable* and actually *running*.
- Occasionally the thread may go from *running* to *dormant*, waiting for some external stimulus; when that arrives, it becomes *runnable* again.
- When completed, a *running* thread becomes *dead*. Data that it holds may persist for other threads to collect.

Some Thread Methods

getPriority and setPriority Every thread has an integer *priority*. In general the scheduler will run a high priority thread in preference to a low priority one.

yield Class method that lets a non-preemptive scheduler switch threads.

sleep Class method that pauses the current thread for a given length of time.

join Waits for a thread to finish, possibly within some time limit.

interrupt Wakes up a dormant thread, raising an InterruptedException in it. If the target thread was running anyway, merely sets a flag that can be checked with Thread.isInterrupted ().

Sharing data and synchronization

When threads share data, they must tread carefully. Suppose two threads sharing an integer a execute a=a+1 and a=a∗2 at the same time. Depending on the scheduler, a may end up with any one of four values, different from one run to the next. The problem is that although Java guarantees that certain *atomic* operations are indivisible, these are quite small. So a=a+1 breaks up as read a; add 1; write a; and between any two the second thread might come in and modify a.

The situation can be even worse in the case of genuinely multiprocessor implementations of Java, since implementations are permitted to make a distinction between *main memory* (where the "official" copy of all the variables is stored) and a separate *working memory* local to each thread. This can lead to a big increase in efficiency, but it also means that changes to a thread's working memory are not instantly reflected in changes to the main memory. There can therefore be a delay (or *latency*) between when a thread makes a change to its working memory and when that change is visible to other threads. In fact, the definition of Java is rather loose with regard to exactly when and how often the main memory needs to be updated to reflect changes to the working memory, and vice versa. As you can imagine, this can lead to a whole host of subtle problems arising from the fact that read/write operations to the memories can happen in different orders.

Fortunately, Java provides mechanisms for ensuring that while a certain bit of code is running, other threads cannot interfere with it. For instance, when we need to be sure that a method on an object is only used by one thread at a time, we declare it to be **synchronized**. Java guarantees that for any given object at most one synchronized method will run at a time. If a second thread tries to call a synchronized method on the same object, then it will *block* (become dormant) until the first thread has finished.

A common example of this is when an object has some private data that must be kept in a consistent state; then all methods that modify the state, or expect to read it consistently, should be marked as synchronized. The Complex class shown below demonstrates this. It encodes a complex number in both rectangular and polar forms, and to make sure that all four instance variables change together both set methods are **synchronized**. The get methods are unsynchronized as each fetches only a single **float** value.

```
class Complex {

  private float x,y,r,theta;

  float getx () { return x; }
  float gety () { return y; }

  float getr () { return r; }
  float gettheta () { return theta; }

  synchronized void setRect (float x, float y) {
    this.x = x; this.y = y;
    r = (float)Math.sqrt(x*x+y*y);
    theta = (float)Math.atan2(y,x);
  }

  synchronized void setPolar (float r, float theta) {
    x = r * (float)Math.cos(theta);
    y = r * (float)Math.sin(theta);
    this.r = r; this.theta = theta;
    }
  }
```

Synchronization gives a method only limited exclusivity: while a synchronized method for an object is running, other non-synchronized methods of the same object can still run concurrently, as can methods synchronized on any other object.

To implement synchronization, every Java object has a *lock* or *mutex* (for "mutual exclusion") which can be held by only one thread at a time. Threads waiting to collect the lock are kept in a queue until it becomes free. How the lock itself is implemented is platform-dependent: Java may use locks provided by the operating system, or take advantage of special test-and-set processor instructions.

If synchronized methods from different objects call each other in a nested fashion, a single thread will acquire several locks. A synchronized method may also call other synchronized methods from the same object, as its thread already holds the lock (Java locks are *reentrant*).

Sometimes we don't need the whole of a method to be run under a lock, just some *critical section* of code. This is done by writing

synchronized (*object*) { *statements* }

which executes the given statements having first obtained the lock attached to the given object. Thus, declaring a whole method to be synchronized amounts to the same thing as protecting the whole of its body by writing **synchronized** (**this**) {...}.

Ian Stark 2002, John Longley 2004