

CHAPTER 1

The Architecture of Threads

PROGRAMMING JAVA THREADS is not nearly as easy (or platform independent) as most books would have you believe, and all Java programs that display a graphical user interface must be multithreaded. This chapter shows you why these statements are true by discussing the architectures of various threading systems and by discussing how those architectures influence how you program threads in Java. Along the way, I'll introduce several key terms and concepts that are not described very well in most intro-to-Java books. Understanding these concepts is essential if you expect to understand the code in the remainder of the book.

The Problems with Threads

Burying your head in the sand and pretending that you don't have to worry about threading issues is a tempting strategy when writing a Java program, but you can't usually get away with it in real production code. Unfortunately, virtually none of the books on Java address threading issues in sufficient depth. If anything, the books go to the opposite extreme, presenting examples that are guaranteed to cause problems in a multithreaded environment as if the code is flawless.

In fact, multithreading is a problem that infuses all your Java code, because you have no way of predicting in exactly what context a particular object or method will be used. Going back after the fact and trying to make non-thread-safe code work in a multithreaded environment is an immensely difficult task. It's best to start out thinking "threads," even if you don't plan to use the code you're writing in a multithreaded way in the future. Unfortunately, there is often a performance penalty to be paid for thread safety, so I can't recommend that *all* code should be thread safe, because paying the penalty can just be too high in some situations. Nonetheless, you should always consider the threading issues when designing the code, even if you end up consciously rejecting thread safety in the implementation.

All Nontrivial Java Programs Are Multithreaded

All Java programs other than simple console-based applications are multithreaded, whether you like it or not. The problem is in Java's Abstract Windowing Toolkit (AWT). (Throughout this book, I'll use "AWT" to mean both the 1.1 AWT library and the *Swing* extensions to AWT as well.) AWT processes operating-system events on a special thread, created by AWT when a program "realizes" (makes visible) its first window. As a consequence, most programs have at least two threads running: the "main" thread, on which `main()` executes, and the AWT thread, which processes events that come in from the operating system and calls any registered listeners in response to those events. It's important to note that all your listener methods run on the AWT thread, not on the main thread (where the listener object is typically created).

There are two main difficulties to this architecture. First, although the listeners run on the AWT thread, they are typically inner-class objects that access an outer-class object that was, in turn, created by (and is accessed by) the main thread. Put another way, listener methods running on AWT thread often access an object that is also manipulated from the main thread—the outer-class object. This is a worst-case synchronization problem, when two threads compete for access to the same object. Proper use of `synchronized` is essential to force the two threads to take turns accessing the object, rather than trying to access it simultaneously.

To make matters worse, the AWT thread that handles the listeners also handles events coming in from the operating system. This means that if your listener methods spend a long time doing whatever they do, OS-level events (such as mouse clicks and key presses) will not be serviced by your program. These events are queued up waiting for service, but they are effectively ignored until the listener method returns. The result is an unresponsive user interface: one that appears to hang. It's immensely frustrating to a user when a program ignores clicks on a Cancel button because the AWT thread has called a listener method that takes forever to execute. (The mouse clicks are ignored until the listener method finishes executing.) Listing 1.1 demonstrates the unresponsive-UI problem. This program creates a frame that holds two buttons labeled "Sleep" and "Hello." The handler for the Sleep button puts the current thread (which will be the Swing event-handler thread) to sleep for five seconds. The Hello button just prints "Hello world" on the console. During the five seconds that elapse after you press the Sleep button, pressing the Hello button has no effect. If you click the Hello button five times, "Hello world" is printed five times as soon as the sleep finishes. The button-press events are queued up while the Swing thread is sleeping, and they are serviced when the Swing thread wakes up.

Listing 1.1: /text/books/threads/ch1/Hang.java

```
01: import javax.swing.*;
02: import java.awt.*;
03: import java.awt.event.*;
04:
05: class Hang extends JFrame
06: {
07:     public Hang()
08:     {   JButton b1 = new JButton( "Sleep" );
09:         JButton b2 = new JButton( "Hello" );
10:
11:         b1.addActionListener
12:         (   new ActionListener()
13:             {   public void actionPerformed((ActionEvent event) )
14:                 {   try
15:                     {   Thread.currentThread().sleep(5000);
16:                         }
17:                     catch(Exception e){}
18:                 }
19:             }
20:         );
21:
22:         b2.addActionListener
23:         (   new ActionListener()
24:             {   public void actionPerformed((ActionEvent event) )
25:                 {   System.out.println("Hello world");
26:                     }
27:             }
28:         );
29:
30:         getContentPane().setLayout( new FlowLayout() );
31:         getContentPane().add( b1 );
32:         getContentPane().add( b2 );
33:         pack();
34:         show();
35:     }
36:
37:     public static void main( String[] args )
38:     {   new Hang();
39:     }
40: }
```

Chapter 1

Many books that discuss java GUI building gloss over both the synchronization and the unresponsive-UI problems. They can get away with ignoring synchronization issues because the trivial examples in those books are often single threaded. That is, 100% of the code in the program is defined inside one or more listeners, all of which are executed serially on the single (AWT) thread. Moreover, the listeners perform trivial tasks that complete so quickly that you don't notice that the UI isn't responding.

In any event, in the real world, this single-threaded approach (doing everything on the AWT thread) just doesn't work. All successful UIs have a few behaviors in common:

- The UI must give you some feedback as an operation progresses. Simply throwing up a box that says "doing such-and-such" is not sufficient. You need to tell the user that progress is being made (a "percent complete" progress bar is an example of this sort of behavior).
- It must be possible to update a window without redrawing the whole thing when the state of the underlying system changes.
- You must provide a way to cancel an in-progress operation.
- It must be possible to switch windows and otherwise manipulate the user interface when a long operation is in progress.

These three rules can be summed up with one rule: **It's not okay to have an unresponsive UI.** It's not okay to ignore mouse clicks, key presses, and so forth when the program is executing a listener method, and it's not okay to do lots of time-consuming work in listeners. The only way to get the reasonable behavior I just described is to use threads. Time-consuming operations must be performed on background threads, for example. Real programs will have many more than two threads running at any given moment.

Java's Thread Support Is Not Platform Independent

Unfortunately, though it's essential to design with threading issues in mind, threads are one of the main places where Java's promise of platform independence falls flat on its face. This fact complicates the implementation of platform-independent multithreaded systems considerably. You have to know something about the possible run-time environments to make the program work correctly in all of them. It is possible to write a platform-independent multithreaded Java program, but you have to do it with your eyes open. This lamentable situation is not really Java's fault; it's almost impossible to write a truly platform-independent threading system. (Doug Schmidt's "Ace" Framework is a good, though complex, attempt. You can get

more information at <http://www.cs.wustl.edu/~schmidt/ACE.html>.) So, before I can talk about hardcore Java programming issues in subsequent chapters, I have to discuss the difficulties introduced by the platforms on which the Java virtual machine (JVM) might run.

Threads and Processes

The first OS-level concept that's important is that of the thread itself (as compared to a *process*). What exactly is a thread (or process), really? It's a data structure deep in the bowels of the operating system, and knowing what's in that data structure can help us answer the earlier question.

The process data structure keeps track of all things memory-related: the global address space, the file-handle table, and so forth. When you swap a process to disk in order to allow another process to execute, all the things in that data structure might have to be staged to disk, including (perhaps) large chunks of the system's core memory. When you think "process," think "memory." Swapping a process is expensive because a lot of memory typically has to be moved around. You measure the context-swap time in seconds. In Java, the process and the virtual machine are rough analogs. All heap data (stuff that comes from `new`) is part of the process, not the thread.

Think of a thread as a thread of execution—a sequence of byte-code instructions executed by the JVM. There's no notion of objects, or even of methods, here. Sequences of instructions can overlap, and they can execute simultaneously. It's commonplace for the same code to be executing simultaneously on multiple threads, for example. I'll discuss all this in more detail later, but think "sequence," not "method."

The *thread* data structure, in contrast to the process, contains the things that it needs to keep track of this sequence. It stores the current machine context: the contents of the registers, the position of the execution engine in the instruction stream, the run-time stack used by methods for local variables and arguments. The OS typically swaps threads simply by pushing the register set on the thread's local stack (inside the thread data structure), putting the thread data structure into some list somewhere, pulling a different thread's data structure off the list, and popping that thread's local stack into the register set. Swapping a thread is relatively efficient, with time measured in milliseconds. In Java, the thread is really a virtual-machine state.

The run-time stack (on which local variables and arguments are stored) is part of the thread data structure. Because multiple threads each have their own run-time stack, the local variables and arguments of a given method are always thread safe. There's simply no way that code running on one thread can access the fields of another thread's OS-level data structure. A method that doesn't access any heap data (any fields in any objects—including `static` ones) can execute simultaneously on multiple threads without any need for explicit synchronization.

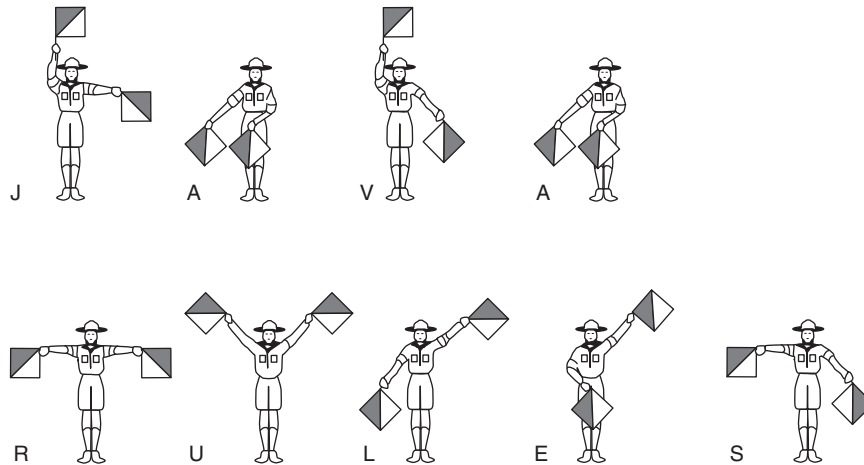
Thread Safety and Synchronization

The phrase *thread safe* is used to describe a method that can run safely in a multi-threaded environment, accessing process-level data (shared by other threads) in a safe and efficient way. The self-contained method described in the previous paragraph is certainly thread safe, but it is really a degenerate case. Thread safety is usually a difficult goal to achieve.

At the center of the thread-safety issue is the notion of *synchronization*—any mechanism that assures that multiple threads:

- start execution at the same time and run concurrently, or
- do not run simultaneously when accessing the same *object*, or
- do not run simultaneously when accessing the same *code*.

I'll discuss ways to do all three of these things in subsequent chapters, but for now, synchronization is achieved by using various objects known collectively as semaphores. A *semaphore* is any object that two threads can use to communicate with one another in order to synchronize their operation. In English, a semaphore is a way to send messages using signalling flags:



Some of you may have learned the semaphore alphabet in the Boy Scouts. Napoleon used the vanes of windmills to send semaphore messages across vast distances; a Java thread uses a semaphore to communicate with another thread. It's not an accident that you are said to *signal* a semaphore (or put it into the *signalled* state)—it's the same metaphor.

Don't be confused by Microsoft documentation that incorrectly applies the word "semaphore" only to a Dijkstra counting semaphore. A semaphore is any of what Microsoft calls "synchronization objects."

Without Java's `synchronized` keyword, you couldn't implement a semaphore in Java, but the `synchronized` keyword alone is not enough. That's not to say that you should throw platform independence out the window and use JNI to call OS-specific synchronization objects; rather, you should build these objects in Java, using the building blocks provided by the language, such as `synchronized`. I'll do just that in subsequent chapters.

Synchronization Is Expensive

One of the main problems with synchronization, whether you use a semaphore or the `synchronized` keyword directly, is overhead. Consider the code in Listing 1.2, which is a simple benchmark meant to demonstrate just how expensive synchronization is. The `test(...)` method (Listing 1.2, line 13) calls two methods 1,000,000 times. One of the methods is synchronized, the other isn't. Results can vary from run to run, but here's a typical output (on a 200MHz P5, NT4/SP3, using JDK ver. 1.2.1 and HotSpot 1.0fcs, build E):

```
% java -verbose:gc Synch
Pass 0: Time lost: 234 ms. 121.39% increase
Pass 1: Time lost: 139 ms. 149.29% increase
Pass 2: Time lost: 156 ms. 155.52% increase
Pass 3: Time lost: 157 ms. 155.87% increase
Pass 4: Time lost: 157 ms. 155.87% increase
Pass 5: Time lost: 155 ms. 154.96% increase
Pass 6: Time lost: 156 ms. 155.52% increase
Pass 7: Time lost: 3,891 ms. 1,484.7% increase
Pass 8: Time lost: 4,407 ms. 1,668.33% increase
```

The `test()` method has to be called several times in order to get the HotSpot JVM to fully optimize the code. That's why the "Pass 0" results seem confusing. This pass takes the most overall time, but the ratio of synchronized to nonsynchronized call time is relatively small because neither method is particularly efficient. Once things settle down (in pass 6), you see that a synchronized call takes about half again as much time to execute as the nonsynchronized variant.

This 1.5-times penalty is significant, but is nothing when compared to passes 7 and 8. The difference is that the earlier passes were all running on a single thread. In the final two passes, two threads are both trying to call the same synchronized method simultaneously, so there is contention. The numbers here are much more significant, with the call to the synchronized method on the order of 150 times less efficient than the nonsynchronized variant. This is a *big* deal. You don't want to synchronize unnecessarily.

*Chapter 1**A Digression*

It's worthwhile explaining what's going on here. The Hotspot JVM typically uses one of two methods for synchronization, depending on whether or not multiple threads are contending for a lock. When there's no contention, an assembly-language atomic-bit-test-and-set instruction is used. This instruction is not interruptible; it tests a bit, sets various flags to indicate the result of the test, then if the bit was not set, it sets it. This instruction is a crude sort of semaphore because when two threads try to set the bit simultaneously, only one will actually do it. Both threads can then check to see if they were the one that set the bit.

If the bit is set (i.e., there is contention), the JVM has to go out to the operating system to wait for the bit to clear. Crossing the interprocess boundary into the operating system is expensive. In NT, it takes on the order of 600 machine cycles just to enter the OS kernel, and this count doesn't include the cycles spent doing whatever you entered the kernel to do. That's why passes 7 and 8 take so much more time, because the JVM must interact with the operating system. Alexander Garthwaite from Sun Labs brought up a few other interesting issues in a recent email to me:

- Synchronized blocks are often different from synchronized methods in that the generated byte code need not properly nest these. As a result, these are often slightly more expensive (particularly in lock-release).
- Some locking strategies use caches of monitors. So, the number and order in which objects are locked can affect performance. More generally, the locking subsystem may use growable structures for various purposes, and these will become more cumbersome to manage as the number of locked objects increases.
- Some locking strategies use a thin-lock/fat-lock strategy. In the thin lock, only simple synchronization is supported and locking depth is often limited to a small number (often somewhere between 16 and 64). Lock inflation occurs when this count is exceeded, when there is contention on the lock, or when a wait or notify operation is performed. Lock deflation can also add costs if it is supported at all.
- For space efficiency in object headers, other information is often either stored in the same word as locking state or it forces lock inflation. A common example is the object's `hashCode()` method. This means that accessing this information in locked objects is often more expensive, and objects with hash codes may be more expensive to lock than ones without.

One other thing that I'll add, if you can reduce the odds of contention, then the locking process is more efficient. This reasoning implies that you should make the synchronization blocks as small as possible so that a given lock will be unlocked most of the time.

Listing 1.2: /text/books/threads/ch1/Synch.java

```
01: import java.util.*;
02: import java.text.NumberFormat;
03:
    /**
    A benchmark to test the overhead of synchronization on a simple
    method invocation. Benchmarking java, particularly when Hot-
    Spot is in the equation, is tricky. There's a good tech note on this
    subject at http://java.sun.com/products/hotspot/Q+A.html.
    */
04: class Synch
05: {
06:     private static long[]    locking_time    = new long[100];
07:     private static long[]    not_locking_time = new long[100];
08:     private static final int  ITERATIONS     = 1000000;
09:
10:     synchronized long locking    (long a, long b){return a + b;}
11:     long              not_locking (long a, long b){return a + b;}
12:
13:     private void test( int id )
14:     {
15:         long start = System.currentTimeMillis();
16:
17:         for(long i = ITERATIONS; --i >= 0 ;)
18:         {   locking(i,i);
19:         }
20:
21:         locking_time[id] = System.currentTimeMillis() - start;
22:         start           = System.currentTimeMillis();
23:
24:         for(long i = ITERATIONS; --i >= 0 ;)
25:         {   not_locking(i,i);
26:         }
27:
28:         not_locking_time[id] = System.currentTimeMillis() - start;
```

Chapter 1

```
29:     }
30:
31:     static void print_results( int id )
32:     {
33:
34:         NumberFormat compositor = NumberFormat.getInstance();
35:         compositor.setMaximumFractionDigits( 2 );
36:
37:         double time_in_synchronization = locking_time[id] - not_locking_time[id];
38:
39:         System.out.println( "Pass " + id + ": Time lost: "
40:             + compositor.format( time_in_synchronization
41:             + " ms. "
42:             + compositor.format( ((double)locking_time[id]/
43:                 not_locking_time[id])*100.0)
44:             + "% increase"
45:             );
46:     }
47:
48:     static public void main(String[] args) throws InterruptedException
49:     {
50:         // First, with no contention:
51:
52:         final Synch tester = new Synch();
53:         tester.test(0); print_results(0);
54:         tester.test(1); print_results(1);
55:         tester.test(2); print_results(2);
56:         tester.test(3); print_results(3);
57:         tester.test(4); print_results(4);
58:         tester.test(5); print_results(5);
59:         tester.test(6); print_results(6);
60:
61:         // Now let's do it again with contention. I'm assuming that
62:         // hotspot has optimized the test method by now, so am only
63:         // calling it once.
64:
65:         final Object start_gate = new Object();
66:
67:         Thread t1 = new Thread()
68:         {
69:             public void run()
70:             {
71:                 try{ synchronized(start_gate) { start_gate.wait(); } }
72:                 catch( InterruptedException e ){ }
73:
74:                 tester.test(7);
75:             }
76:         };
77:
78:         Thread t2 = new Thread()
```

```
75:      { public void run()
76:          { try{ synchronized(start_gate) { start_gate.wait(); } }
77:            catch( InterruptedException e ){ }
78:
79:            tester.test(8);
80:          }
81:      };
82:
83:      Thread.currentThread().setPriority( Thread.MIN_PRIORITY );
84:
85:      t1.start();
86:      t2.start();
87:
88:      synchronized(start_gate){ start_gate.notifyAll(); }
89:
90:      t1.join();
91:      t2.join();
92:
93:      print_results( 7 );
94:      print_results( 8 );
95:  }
96: }
```

Avoiding Synchronization

Fortunately, explicit synchronization is often avoidable. Methods that don't use any of the state information (such as fields) of the class to which they belong don't need to be synchronized, for example. (That is, they use only local variables and arguments—no class-level fields—and they don't modify external objects by means of references that are passed in as arguments.) There are also various class-based solutions, which I discuss in subsequent chapters (such as the synchronization wrappers used by the Java collection classes).

You can sometimes eliminate synchronization simply by using the language properly, however. The next few sections show you how.

Chapter 1

Atomic Energy: Do Not Synchronize Atomic Operations

The essential concept vis-a-vis synchronization is *atomicity*. An “atomic” operation cannot be interrupted by another thread, and naturally atomic operations do not need to be synchronized.

Java defines a few atomic operations. In particular, assignment to variables of any type except `long` and `double` is atomic. To understand ramifications of this statement, consider the following (hideously non-object-oriented) code:

```
class Unreliable
{   private long x;

    public long get_x(           ){ return x;   }
    public void set_x(long value ){ x = value;  }
}
```

Thread one calls:

```
obj.set_x( 0 );
```

A second thread calls:

```
obj.set_x( 0x123456789abcdef );
```

The problem is the innocuous statement:

```
x = value;
```

which is effectively treated by the JVM as two separate 32-bit assignments, not a single 64-bit assignment:

```
x.high_word = value.high_word;
x.low_word  = value.low_word;
```

Either thread can be interrupted by the other halfway through the assignment operation—after modifying the high word, but before modifying the low word. Depending on when the interruption occurs (or if it occurs), the possible values of `x` are `0x0123456789abcdef`, `0x0123456700000000`, `0x0000000089abcdef`, or `0x0000000000000000`. There’s no telling which one you’ll get. The only way to fix this problem is to redefine both `set_x()` and `get_x()` as synchronized or wrap the assignment in a synchronized block.

The `volatile` Keyword

Another keyword of occasional interest is `volatile`. The issue here is not one of synchronization, but rather of optimization. If one method sets a flag, and another tests it, the optimizer might think that the value of the flag never changes and optimize the test out of existence. Declaring the variable as `volatile` effectively tells the optimizer not to make any assumptions about the variable's state. In general, you'll need to use `volatile` only when two threads both access a public flag, something that shouldn't happen in well-crafted OO systems. In any event, for reasons that I don't want to go into here, `volatile` can behave in unpredictable ways on multiprocessor machines. Until the Java language specification is fixed, it's best to use explicit synchronization to avoid these problems.

Fortunately, this problem doesn't arise with 32-bit (or smaller) variables. That is, if all that a method does is set or return a value, and that value is *not* a long or double, then that method doesn't have to be synchronized. Were the earlier `x` redefined as an `int`, no synchronization would be required.

Bear in mind that only assignment is guaranteed to be atomic. A statement like `x+=y` (or `x+=y`) is *never* thread safe, no matter what size `x` and `y` are. You could be preempted after the increment but before the assignment. You must use the `synchronized` keyword to get atomicity in this situation.

Race Conditions

Formally, the sort of bug I just described—when two threads simultaneously contend for the same object and, as a consequence, leave the object in an undefined state—is called a *race condition*. Race conditions can occur anywhere that any sequence of operations must be atomic (not preemptable), and you forget to make them atomic by using the `synchronized` keyword. That is, think of `synchronized` as a way of making complex sequences of operations atomic, in the same way that assignment to a `boolean` is atomic. The `synchronized` operation can't be preempted by another thread that's operating on the same data.

Immutability

An effective language-level means of avoiding synchronization is immutability. An *immutable* object is one whose state doesn't change after it's created. A Java `String` is a good example—there's no way to modify the `String` once it's created. (The expression `string1 += string2` is actually treated like `string1 = string1 + string2`; a third `string` is created by concatenating the two operands, then the target is overwritten to reference this third string. As usual, this operation is not atomic.)

Chapter 1

Since the value of an immutable object never changes, multiple threads can safely access the object simultaneously, so no synchronization is required.

Create an immutable object by making *all* of the fields of a class `final`. The fields don't all have to be initialized when they are declared, but if they aren't they *must* be explicitly initialized in every constructor. For example:

```
class I_am_immutable
{   private final int MAX_VALUE = 10;
    private final int blank_final;

    public I_am_immutable( int initial_value )
    {   blank_final = initial_value;
    }
}
```

A `final` field that's initialized by the constructor in this way is called a *blank final*.

In general, if you are accessing an object a lot, but not modifying it much, making it immutable is a good idea since none of the methods of the object's class need to be synchronized. If you modify the object a lot, however, the overhead of copying the object will be much higher than the overhead of synchronization, so an immutable-object approach doesn't make sense. Of course, there is a vast gray area where neither approach is obviously better.

Synchronization Wrappers

Often it's the case that you need synchronization sometimes, but not all the time. A good example is the Java 2 Collection classes. Typically, collections will be accessed from within synchronized methods, so it would be contraindicated for the methods of the collection to be synchronized, since you'd be unnecessarily acquiring two locks (the one on the object that used the collection and the other on the collection itself). Java's solution to this problem is generally applicable: use a synchronization wrapper. The basic notion of the Gang-of-Four *Decorator* design pattern is that a Decorator both implements some interface and also contains an object that implements the same interface. (The "Gang-of-Four" referenced in the previous sentence are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the authors of the excellent book *Design Patterns: Elements of Reusable Object-Oriented Software* [Reading: Addison Wesley, 1995].) The container implements the same methods as the contained object, but modifies the behavior of the method as it passed the request through to the contained object. The classes in the `java.io` package are all Decorators: A `BufferedInputStream` both implements `InputStream` and contains an instance of some `InputStream`—you talk to the contained object through the container, which modifies the behavior of the contained object. (The `read()` method buffers characters in the `BufferedInputStream` decorator and doesn't

buffer in the contained `FileInputStream`. The `BufferedInputStream` container gets its characters from the contained `FileInputStream` object.)

You can put this technique to use to provide synchronization on an as-needed basis. For example:

```
interface Some_interface
{
    Object message();
}

class Not_thread_safe implements Some_interface
{
    public Object message()
    {
        // ... Implementation goes here
        return null;
    }
}

class Thread_safe_wrapper implements Some_interface
{
    Some_interface not_thread_safe;

    public Thread_safe_wrapper( Some_interface not_thread_safe )
    {
        this.not_thread_safe = not_thread_safe;
    }

    public Some_interface extract()
    {
        return not_thread_safe;
    }

    public synchronized Object message()
    {
        return not_thread_safe.message();
    }
}
```

When thread safety isn't an issue, you can just declare and use objects of class `Not_thread_safe` without difficulty. When you need a thread-safe version, just wrap it:

```
Some_interface object = new Not_thread_safe();
//...

object = new Thread_safe_wrapper(object); // object is now thread safe
```

when you don't need thread-safe access any more, unwrap it:

```
object = ((Thread_safe_wrapper)object).extract();
```

Chapter 1

Concurrency, or How Can You Be Two Places at Once (When You're Really Nowhere at All)

The next OS-related issue (and the main problem when it comes to writing platform-independent Java) has to do with the notions of *concurrency* and *parallelism*. Concurrent multithreading systems give the appearance of several tasks executing at once, but these tasks are actually split up into chunks that share the processor with chunks from other tasks. Figure 1.1 illustrates the issues. In parallel systems, two tasks are actually performed simultaneously. Parallelism requires a multiple-CPU system.

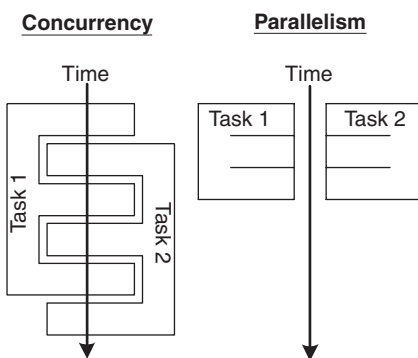


Figure 1.1. Concurrency vs. Parallelism

Multiple threads don't necessarily make your program faster. Unless you're spending a lot of time blocked, waiting for I/O operations to complete, a program that uses multiple concurrent threads will often run slower than an equivalent single-threaded program (although it will often be better organized than the equivalent single-thread version). A program that uses multiple threads running in parallel on multiple processors will run much faster, of course. If speed is important, a multithreaded program should have no more threads running at any given moment than there are processors in the system. More threads can exist in this program, but they should be suspended, waiting for some event to occur.

The main reason that Java's threading system isn't platform independent is that parallelism is impossible unless you use the underlying operating system's threading model. Java, at least in theory, permits threading to be simulated entirely by the JVM, thereby avoiding the time penalty for entering the OS kernel that I discussed earlier. This approach precludes any parallelism in your application, however: If no operating-system-level threads are used, the OS looks at the JVM instance as a single-threaded application, which will be scheduled to a single processor. The net result would be that no two Java threads running under the same JVM instance would ever run in parallel, even if you had multiple CPUs and your JVM was the only process that was active. Two instances of the JVM running separate applications could run in parallel, of course, but I want to do better than that. To get parallelism, the JVM *must* map Java threads through to operating-system threads. Unfortunately, different operating systems implement threads in different ways; so, you can't afford to ignore the differences between the various threading models if platform independence is important.

Get Your Priorities Straight

I'll demonstrate the ways that all the issues I just discussed can impact your programs by comparing two operating systems: Solaris and Windows NT.

Java, in theory at least, provides ten priority levels for threads. (If two or more threads are both waiting to run, the one with the highest priority level will execute.) In Solaris, which supports 2^{31} priority levels, having ten levels is no problem. You give up a lot of fine control over priority by restricting yourself to one of these ten levels, but everything will work the way that you expect.

NT, on the other hand, has at most seven priority levels available, which have to be mapped into Java's ten. This mapping is undefined, so lots of possibilities present themselves. (Java priority levels 1 and 2 might both map to NT priority-level 1, and Java priority levels 8, 9, and 10 might all map to NT level 7, for example. Other combinations, such as using only five of the available levels and mapping pairs of Java levels to a single NT level, are also possible). NT's paucity of priority levels is a problem if you want to use priority to control scheduling.

Things are made even more complicated by the fact that NT priority levels are not fixed. NT provides a mechanism called "priority boosting," which you can turn off with a C system call, but not from Java. When priority boosting is enabled, NT boosts a thread's priority by an indeterminate amount for an indeterminate amount of time every time it executes certain I/O-related system calls. In practice, this means that a thread's priority level could be higher than you think because that thread happened to perform an I/O operation at an awkward time. The point of the priority boosting is to prevent threads that are doing background processing from impacting the apparent responsiveness of UI-heavy tasks. Other operating systems have more-sophisticated algorithms that typically lower the priority of background processes. The down side of this scheme, particularly when implemented on a per-thread rather than per-process level, is that it's very difficult to use priority to determine when a particular thread will run.

It gets worse.

In Solaris—as is the case in all Unix systems and every contemporary operating system that I know of *except* the Microsoft operating systems—processes have priority as well as threads. The threads of high-priority processes can't be interrupted by the threads of low-priority processes. Moreover, the priority level of a given process can be limited by a system administrator so that a user process won't interrupt critical OS processes or services. NT supports none of this. An NT process is just an address space. It has no priority per se and is not scheduled. The system schedules threads; then, if that thread is running under a process that is not in memory, the process is swapped in. NT thread priorities fall into various "priority classes," that are distributed across a continuum of actual priorities. The system is shown in Figure 1.2.

The columns are actual priority levels, only twenty-two of which must be shared by all applications. (The others are used by NT itself.) The rows are priority classes.

Chapter 1

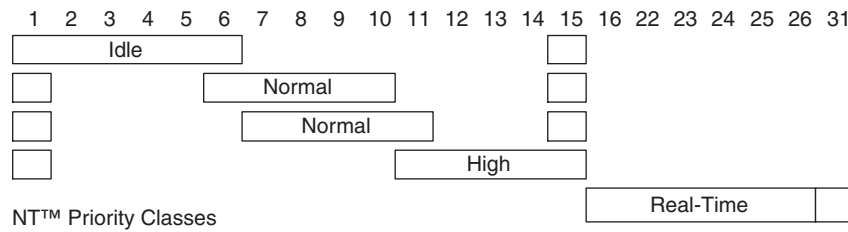


Figure 1-2. Windows NT's Priority Achitecture

The threads running in a process pegged at the “Idle” priority class are running at levels 1–6 and 15, depending on their assigned logical priority level. The threads of a process pegged as “Normal” priority class will run at levels 1, 6–10, or 15 if the process doesn't have the input focus. If it does have the input focus, the threads run at levels 1, 7–11, or 15. This means that a high-priority thread of an idle-priority-class process can preempt a low-priority thread of a normal-priority-class process, but only if that process is running in the background. Notice that a process running in the “High” priority class only has six priority levels available to it. The other classes have seven.

NT provides no way to limit the priority class of a process. Any thread on any process on the machine can take over control of the box at any time by boosting its own priority class, and there's no defense. Solaris, on the other hand, does support the notion of process priority precisely because you need to prevent screen savers from interfering with system-critical tasks. A high-priority process simply shouldn't be preempted by a low-priority process, particularly in a server. I guess the good people at Microsoft didn't think that anyone would *really* be using NT as a server operating system. Anyway, the technical term I use to describe NT's priority is “unholy mess.” In practice, priority is virtually worthless under NT.

So what's a programmer to do? Between NT's limited number of priority levels and its uncontrollable priority boosting, there's no absolutely safe way for a Java program to use priority levels for scheduling. One workable compromise is to restrict yourself to `Thread.MAX_PRIORITY`, `Thread.MIN_PRIORITY`, and `Thread.NORM_PRIORITY` when you call `setPriority()`. This restriction at least avoids the ten-mapped-to-seven-levels problem. I suppose you could use the `os.name` system property to detect NT, and then call a native method to turn off priority boosting, but that won't work if your app is running under Internet Explorer unless you also use Sun's JVM plug-in. (Microsoft's JVM uses a nonstandard native-method implementation.) In any event, I hate to use native methods. I usually avoid the problem as much as possible by putting most threads at `NORM_PRIORITY` and using scheduling mechanisms other than priority. (I'll discuss some of these in subsequent chapters.)

Cooperate!

There are typically two threading models supported by operating systems: cooperative and preemptive.

The Cooperative Multithreading Model

In a *cooperative* system, a thread retains control of its processor until it decides to give it up (which might be never). The various threads have to cooperate with each other or all but one of the threads will be *starved* (never given a chance to run). Scheduling in most cooperative systems is done strictly by priority level. When the current thread gives up control, the highest-priority waiting thread gets control. (An exception to this rule is Windows 3.x, which uses a cooperative model but doesn't have much of a scheduler. The window that has the focus gets control.)

The main advantage of cooperative multithreading is that it's very fast and has a very low overhead when compared to preemptive systems. For example, a *context swap*—a transfer of control from one thread to another—can be performed entirely by a user-mode subroutine library without entering the OS kernel (which costs 600 machine cycles in NT). A user-mode context swap in a cooperative system does little more than a C `setjump/longjump` call would do. You can have thousands of cooperative threads in your applications without significantly impacting performance. Because you don't lose control involuntarily in cooperative systems, you don't have to worry about synchronization either. Just don't give up control until it's safe to do so. You never have to worry about an atomic operation being interrupted. The two main disadvantages of the cooperative model are:

1. It's very difficult to program cooperative systems. Lengthy operations have to be manually divided into smaller chunks, which often must interact in complex ways.
2. The cooperative threads can never run in parallel.

The Preemptive Multithreading Model

The alternative to a cooperative model is a *preemptive* one, where some sort of timer is used by the operating system itself to cause a context swap. That is, when the timer “ticks” the OS can abruptly take control away from the running thread and give control to another thread. The interval between timer ticks is called a *time slice*.

Preemptive systems are less efficient than cooperative ones because the thread management must be done by the operating-system kernel, but they're easier to program (with the exception of synchronization issues) and tend to be more reliable

Chapter 1

because starvation is less of a problem. The most important advantage to preemptive systems is parallelism. Because cooperative threads are scheduled by a user-level subroutine library, not by the OS, the best you can get with a cooperative model is concurrency. To get parallelism, the OS must do the scheduling. Four threads running in parallel on four processors will run more than four times faster than the same four threads running concurrently (because there is no context-swap overhead).

Some operating systems, like Windows 3.1, only support cooperative multi-threading. Others, like NT, support only preemptive threading. (You can simulate cooperative threading in NT with a user-mode library. NT has such a library called the “fiber” library, but fibers are buggy, and aren’t fully integrated into the operating system.) Solaris provides the best (or worst) of all worlds by supporting both cooperative and preemptive models in the same program. (I’ll explain this in a moment.)

Mapping Kernel Threads to User Processes

The final OS issue has to do with the way in which kernel-level threads are mapped into user-mode processes. NT uses a one-to-one model, illustrated in Figure 1.3.

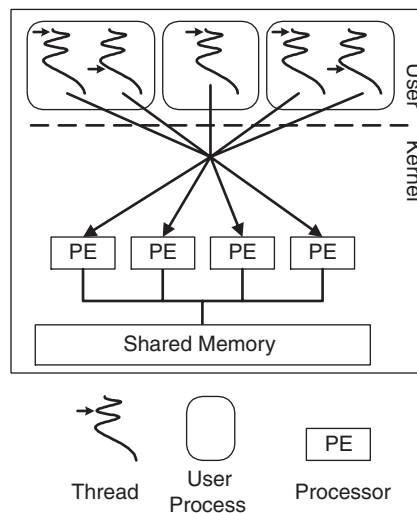


Figure 1.3. The NT Threading Model

NT user-mode threads effectively *are* kernel threads. They are mapped by the OS directly onto a processor and they are always preemptive. All thread manipulation and synchronization are done via kernel calls (with a 600-machine-cycle overhead for every call). This is a straightforward model, but is neither flexible nor efficient.

The Solaris model in Figure 1.4 is more interesting. Solaris adds *lightweight process* (LWP) to the notion of a thread. The LWP is a schedulable unit on which one or more threads can run. Parallel processing is done on the LWP level. Normally, LWPs reside in a pool, and they are assigned to particular processors as necessary. An LWP can be *bound* to a specific processor if it's doing something particularly time critical, however, thereby preventing other LWPs from using that processor.

Up at the user level, you have a system of cooperative, or *green* threads. In a simple situation, a process will have one LWP shared by all of the green threads. The threads must yield control to each other voluntarily, but the single LWP that the threads share can be preempted by an LWP in another process. This way the processes are preemptive with respect to each other (and can execute in parallel), but the threads within the process are cooperative (and execute concurrently).

A process is not limited to a single LWP, however. The green threads can share a pool of LWPs in a single process. The green threads can be attached (or bound) to an LWP in two ways:

1. The programmer explicitly “binds” one or more threads to a specific LWP. In this case, the threads sharing a LWP must cooperate with each other, but they can preempt (or be preempted by) threads bound to a different LWP. If every green thread was bound to a single LWP, you'd have an NT-style preemptive system.
2. The threads are bound to green threads by the user-mode scheduler. This is something of a worst case from a programming point of view because you can't assume a cooperative or a preemptive environment. You may have to yield to other threads if there's only one LWP in the pool, but you might also be preempted.

The Solaris threading model gives you an enormous amount of flexibility. You can choose between an extremely fast (but strictly concurrent) cooperative system, a slower (but parallel) preemptive system, or any combination of the two. But (and this is a *big* “but”) none of this flexibility is available to you, the hapless Java programmer, because you have no control over the threading model used by the JVM. For example, early versions of the Solaris JVM were strictly cooperative. Java threads were all green threads sharing a single LWP. The current version of the Solaris JVM uses multiple LWPs and no green threads at all.

So why do you care? You care precisely because you have no control—you have to program as if all the possibilities might be used by the JVM. In order to write platform-independent code, you must make two seemingly contradictory assumptions:

1. You can be preempted by another thread at any time. You must use the `synchronized` keyword carefully to assure that nonatomic operations work correctly.

Chapter 1

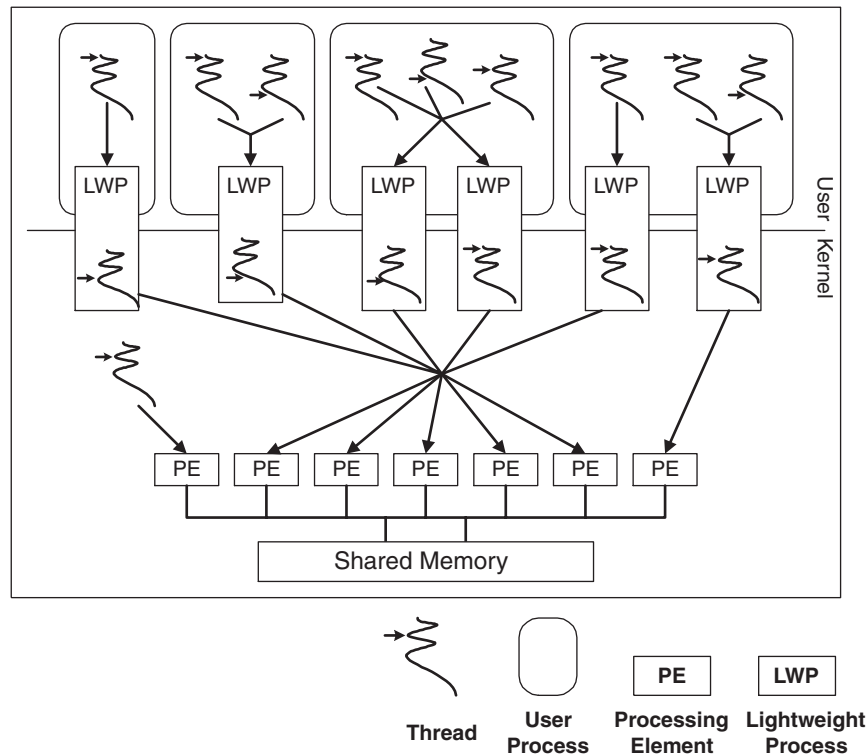


Figure 1.4. The Solaris Threading Model

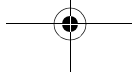
2. You will never be preempted unless you give up control. You must occasionally perform some operation that will give control to other threads so that they can have a chance to run. Use `yield()` and `sleep()` in appropriate places (or make blocking I/O calls). For example, you might want to consider calling `yield()` every 100 iterations or so of a long loop, or voluntarily going to sleep for a few milliseconds every so often to give lower-priority threads a chance to run. (The `yield()` method will yield control only to threads running at your priority level or higher).

Wrapping Up

So those are the main OS-level issues that you have to consider when you're writing a Java program. Since you can make no assumptions about your operating environment, you have to program for the worst case. For example, you have to assume that you can be preempted at any time, so you must use `synchronized` appropriately, but you must also assume that you will never be preempted, so you must also



occasionally use `yield()`, `sleep()`, or blocking I/O calls to permit other threads to run. Any use of priority is problematic: You can't assume that two adjacent priority levels are different. They might not be after NT has mapped Java's ten levels into its seven levels. Similarly, you can't assume that a priority-level-two thread will always be higher priority than one that runs at level 1—it might not be if NT has “boosted” the priority level of the lower-priority thread.





<http://www.springer.com/978-1-893115-10-1>

Taming Java Threads

Holub, A.

2000, X, 300 p. 104 illus., Softcover

ISBN: 978-1-893115-10-1

A product of Apress