**Chapter 1: The Architecture of Threads**

**The Problems with Threads**

AWT library and 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).

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.

**Java's Thread Support Is Not Platform Independent**

**Threads and Processes**

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 contextswap

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 - "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.

**Thread Safety and Synchronization**

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*.

A *semaphore* is any object that two threads can use to communicate with one another in order to synchronize their operation.

Without Java's synchronized keyword, you couldn't implement a semaphore in Java, but the

synchronized keyword alone is not enough.

**Synchronization Is Expensive**

|  |  |
| --- | --- |
| 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 | Pass 0: Time lost: 226 ms. 2,611.11% increase  Pass 1: Time lost: 230 ms. 3,933.33% increase  Pass 2: Time lost: 227 ms. 3,883.33% increase  Pass 3: Time lost: 222 ms. 2,566.67% increase  Pass 4: Time lost: 225 ms. 3,314.29% increase  Pass 5: Time lost: 222 ms. 2,875% increase  Pass 6: Time lost: 223 ms. 3,285.71% increase  Pass 7: Time lost: 1,177 ms. 16,914.29% increase  Pass 8: Time lost: 1,173 ms. 14,762.5% increase |

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.

**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.

**Avoiding Synchronization**

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.

**Atomic Energy: Do Not Synchronize Atomic Operations**

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.

Fortunately, this problem doesn't arise with 32-bit (or smaller) variables.

**The volatile Keyword**

The issue here is not one of synchronization, but rather of optimization.

Declaring the variable as volatile effectively tells the optimizer not to make any assumptions about the variable's state.

**Race Conditions**

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,

**Immutability**

An *immutable* object is one whose state doesn't change after it's created.

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.

**Synchronization Wrappe**

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.

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

**Concurrency, or How Can You Be Two Places at Once (When You're**

**Really Nowhere at All)**



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.

In parallel systems, two tasks are actually performed simultaneously. Parallelism requires a multiple-CPU

system.

Multiple threads don't necessarily make your program faster.

A program that uses multiple threads running in parallel on multiple processors will run much faster, of course.

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…

If no operating-system-level threads are used, the OS looks at the JVM instance as a singlethreaded 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…

To get parallelism, the JVM *must* map Java threads through to operating-system threads.

**Get Your Priorities Straight**

Solaris and Windows NT 2^31 priorities versus 7 in NT…

**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).

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.

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).

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 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).

**Windows 3.1**, only support **cooperative** multithreading…

**NT**, support only **preemptive** threading.

Solaris provides the best (or worst) of all worlds by supporting **both** cooperative and preemptive models in the same program.

**Mapping Kernel Threads to User Processes**



**Figure 1.3:** The NT Threading Model



**Figure 1.4:** The Solaris Threading Model

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.

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.
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).

**Chapter 2: The Perils of Multithreaded**

**Programming**

**Monitors and Exclusion Semaphores (Mutex)**