**Thread Scheduling**

The key to understanding Java thread scheduling is to realize that a CPU is a scarce resource. When two or more threads want to run on a single-processor machine, they end up competing for the CPU, and it’s up to someone –either the programmer, the Java virtual machine, or the operating system –to make sure that the CPU is shared among these threads. The same is true whenever a program has more threads than the machine hosting the program has CPUs.

**The Scheduling Process**

At a conceptual level, every thread in the Java virtual machine can be in one of four states:

**Initial**

A thread object is in the initial state from the period when it is created (that is, when its constructor is called) until the start() method of the thread object is called.

**Runnable**

A thread is in the runnable state once its start() method has been called. A thread leaves the runnable state in various ways, but the runnable state can be thought of as a default state: if a thread isn’t in any other state, it’s in the runnable state.

A thread that is in the runnable state may not actually be running; it may be waiting for a CPU. A thread that is running on a CPU is called a currently running thread.

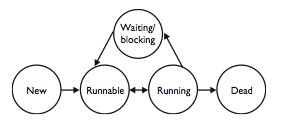
**Blocked**

A thread that is blocked is one that cannot be run because it is waiting for some specific event to occur. Threads block for many reasons: they attempt to read data (e.g., from a socket) when no data is available; they execute a thread blocking method (e.g., the sleep(), wait(), or join() methods); or they attempt to acquire a synchronization lock that another thread already holds. We’ve seen APIs that also block, but internally those methods are all executing the wait() method.

**Exiting**

A thread is in the exiting state once its run() method returns (or its deprecated stop() method has been called).

The basic process of thread scheduling is essentially the same whether it’s performed by the Java virtual machine or the underlying operating system.



**Popular threading Implementations**

Green threads

This is the simplest model. In this model, the operating system doesn’t know anything about Java threads at all; it is up to the virtual machine to handle all the details of the threading API. From the perspective of the operating system, there is a single process and a single thread.

Each thread in this model is an abstraction within the virtual machine: the virtual machine must hold within the thread object all information related to the thread. This includes the thread’s stack, a program counter that indicates which Java instruction the thread is executing, and other bookkeeping information about the thread. The virtual machine is the responsible for switching thread contexts: that is, saving this information for one particular thread, loading it from another thread, and then executing the new thread. As far as the operating system is concerned, the virtual machine is just executing arbitrary code; the fact that the code is emulating many different threads is unknown outside of the virtual machine.

This model is known in Java as the green thread model. In other circles, these threads are often called user-level threads because they exist only within the user level of the application: no calls into the operating system are required to handle any thread details.

In the early days of Java, the green thread model was fairly common, particularly on most Unix platforms. Some specialized operating systems today use this model, but most computers use a native, system-level model.

The green thread model is completely deterministic with respect to scheduling.

**Windows Native Threads**

In the native-threading model used on 32-bit Windows operating system, the OS is fully cognizant of the multiple threads that the virtual machine uses, and there is a one-to-one mapping between Java threads and operating system threads. Therefore, the scheduling of Java threads is subject to the underlying scheduling of threads by the operating system.

**Solaris Native Threads**

Recent versions of the Solaris Operating Environment have had two different threading models. Solaris 7 featured a complex, two-level threading system, with user-level threads and system-level lightweight processes (LWPs). Java threads were equivalent to Solaris user-level threads, and there is an M-to-N mapping between the user-level threads and LWPs. Much of the flexibility of this model is lost on the Java developer, who can directly influence only the priority (and number) of the user-level threads but not the underlying LWPs.

In Solaris 9, a new one-to-one threading model is used. That makes it conceptually similar to the models on Windows operating systems, though its implementation details are quite different.

**Linux Native Threads**

Until JDK 1.3, Linux-based virtual machines tended to use a green thread model. Some used Linux's native threads, but the kernel support for those threads did not support a large number of concurrent threads. JDK 1.3 added support for Linux native threads. However, the Linux kernel at the time was not optimal for threaded applications; in particular, the ps command listed all threads as if they were different processes.

New Linux kernels use the Native Posix Thread Library (NPTL), which provides the same one-to-one mapping of Java threads to kernel threads that we've seen in other operating systems. The complex priority calculation for those threads is similar to what we saw on Solaris, where the Java priority is only a small factor in the calculation. JDK 1.4.2 is the first version of Java to support this new kernel.