CPU Scheduling

This presents an overview of scheduling in the Java virtual machine, as well as how Java threads are scheduled on the host operating system.

6.1 Overview

We now consider scheduling in Java. The specification for the JVM has a loosely defined scheduling policy simply stating that each thread has a priority and that higher-priority threads will run in preference to threads with lower priorities. However, unlike the case with strict priority-based scheduling, it is possible that a lower-priority thread may get an opportunity to run at the expense of a higher-priority thread. Since the specification does not say that a scheduling policy must be preemptive, it is possible that a thread with a lower priority may continue to run even as a higher-priority thread becomes runnable.

Furthermore, the specification for the JVM does not indicate whether or not threads are time-sliced using a round-robin scheduler (Section Section 6.3.4)—that decision is up to the particular implementation of the JVM. If threads are time-sliced, then a runnable thread executes until one of the following events occurs:

1. Its time quantum expires.
2. It blocks for I/O.
3. It exits its `run()` method.

On systems that support preemption, a thread running on a CPU may also be preempted by a higher-priority thread.

So that all threads have an equal amount of CPU time on a system that does not perform time slicing, a thread may yield control of the CPU with the `yield()` method. By invoking the `yield()` method, a thread *suggests* that it is willing to relinquish control of the CPU, allowing another thread an opportunity to run. This yielding of control is called **cooperative multitasking**. The use of the `yield()` method appears as

```
public void run() {
   while (true) {
      // perform a CPU-intensive task
         . . .
      // now yield control of the CPU
      Thread.yield();
   }
}
```

6.2 Thread Priorities

Each Java thread is assigned a priority that is a positive integer within a given range. A thread is given a default priority when it is created. Unless it is changed explicitly by the program, it maintains the same priority throughout its lifetime; the JVM does not dynamically alter priorities. The Java `Thread` class identifies the following thread priorities:

Priority	Comment
`Thread.MIN_PRIORITY`	The minimum thread priority
`Thread.MAX_PRIORITY`	The maximum thread priority
`Thread.NORM_PRIORITY`	The default thread priority

`MIN_PRIORITY` has a value of 1; `MAX_PRIORITY`, a value of 10; and `NORM_PRIORITY`, a value of 5. Every Java thread has a priority that falls somewhere within this range. (There is in fact a priority 0, but it is reserved for threads created by the JVM; developers cannot assign a thread with priority 0.) The default priority is `NORM_PRIORITY`.

When a thread is created, it is given the same priority as the thread that created it. The priority of a thread can also be set explicitly with the `setPriority()` method. The priority can be set either before the thread is started or while the thread is active. The class `HighThread` (Figure 6.1) illustrates using the `setPriority()` method to increase the priority of the thread by 3.

Because the JVM is typically implemented on top of a host operating system, the priority of a Java thread is related to the priority of the kernel thread to which it is mapped. As might be expected, this relationship varies from

```
public class HighThread implements Runnable
{
 public void run() {
  Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 3);
  // remainder of run() method
   . . .
 }
}
```

Figure 6.1 Setting a priority using `setPriority()`.

Java priority	Win32 priority
1 (MIN_PRIORITY)	LOWEST
2	LOWEST
3	BELOW_NORMAL
4	BELOW_NORMAL
5 (NORM_PRIORITY)	NORMAL
6	ABOVE_NORMAL
7	ABOVE_NORMAL
8	HIGHEST
9	HIGHEST
10 (MAX_PRIORITY)	TIME_CRITICAL

Figure 6.2 The relationship between the priorities of Java and Win32 threads.

system to system. Thus, changing the priority of a Java thread through the `setPriority()` system call can have different effects depending on the host operating system.

For example, consider Solaris systems. As described in Section Section 6.7.3, Solaris threads are assigned a priority between 0 and 59. On these systems, the ten priorities assigned to Java threads must somehow be associated with the sixty possible priorities of the kernel threads. The priority of the kernel thread to which a Java thread is mapped is based on a combination of the Java-level priority and the dispatch table shown in Figure 6.24.

Win32 systems identify seven priority levels. As a result, different Java thread priorities may map to the same priority of a kernel thread. For example, `Thread.NORM_PRIORITY` +1 and `Thread.NORM_PRIORITY` +2 map to the same kernel priority; altering the priority of Java threads on Win32 systems may have no effect on how such threads are scheduled. The relationship between the priorities of Java and Win32 threads is shown in Figure 6.2.

It is important to note that the specification for the JVM does not identify how a JVM must implement thread priorities. JVM designers can implement prioritization however they choose. In fact, a JVM implementation may choose to ignore calls to `setPriority()` entirely.

6.3 Java Thread Scheduling on Solaris

Let's look more closely at Java thread scheduling on Solaris systems. The relationship between the priorities of a Java thread and its associated kernel thread has an interesting history. On Solaris, each Java thread is assigned a unique user thread. Furthermore, user threads are mapped to kernel threads through a lightweight process, or LWP, as illustrated in Figure 6.3. Solaris uses the `priocntl()` and `thr_setprio()` system calls to change the priorities of LWPs and user threads, respectively. The kernel is aware of priority changes

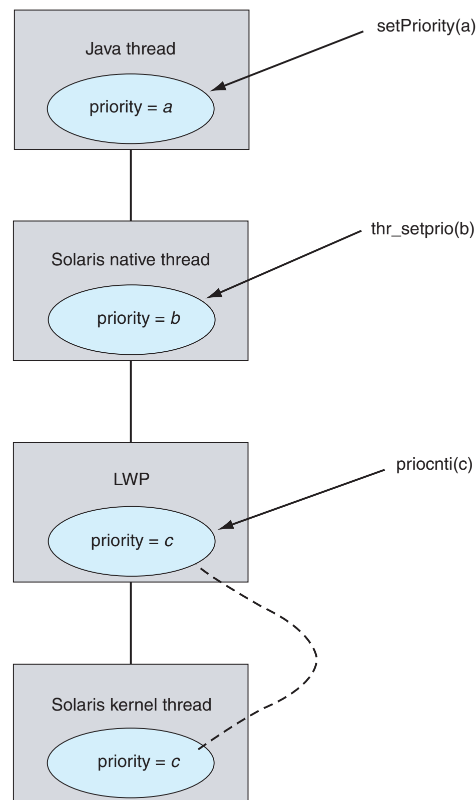

Figure 6.3 User threads are mapped to kernel threads through a lightweight process.

made to the LWP with `priocntl()`, but changes made to the priority of a user thread with `thr_setprio()` are not reflected in the kernel.

Prior to Solaris 9, Solaris used the many-to-many model of mapping user and kernel threads (Section Section 4.3.3). The difficulty with this model was that, although it was possible to change the priority of the LWP to which a Java thread was mapped, this mapping was dynamic, and the Java thread could switch to another LWP without the JVM's awareness. This issue was addressed when Solaris 9 adopted the one-to-one model (Section Section 4.3.2), thereby ensuring that a Java thread is assigned to the same LWP for its lifetime.

When Java's `setPriority()` method was called prior to Version 1.4.2 of the JVM on Solaris, the JVM would change only the priority of the user thread, using the `thr_setprio()` system call; the priority of the underlying LWP was not changed, as the JVM did not invoke `priocntl()`. (This was because Solaris used the many-to-many threading model at that time. Changing the priority of the LWP made little sense, as a Java thread could migrate between different LWPs without the kernel's knowledge.) However, by the time Version 1.4.2 of the JVM appeared, Solaris had adopted the one-to-one model. As a result, when a Java thread invoked `setPriority()`, the JVM would call both `thr_setprio()` and `priocntl()` to alter the priority of the user thread and the LWP, respectively.

Version 1.5 of the JVM on Solaris addressed issues concerning the relative priorities of Java threads and threads running in native C and C++ programs.

On Solaris, by default, a C or C++ program initially runs at the highest priority in its scheduling class. However, the default priority given to a Java thread at `Thread.NORM_PRIORITY` is in the middle of the priority range for its scheduling class. As a result, when Solaris concurrently ran both a C and a Java program with default scheduling priorities, the operating system typically favored the C program. Version 1.5 of the JVM on Solaris assigns Java priorities from `Thread.NORM_PRIORITY` to `Thread.MAX_PRIORITY`, the highest priority in the scheduling class. Java priorities `Thread.MIN_PRIORITY` to `Thread.NORM_PRIORITY` −1 are assigned correspondingly lower priorities. The advantage of this scheme is that Java programs now run with priorities equal to those of C and C++ programs. The disadvantage is that changing the priority of a Java thread from `setPriority(Thread.NORM_PRIORITY)` to `setPriority(Thread.MAX_PRIORITY)` has no effect.

6.4 Scheduling Features in Java 1.5

With such a loosely defined scheduling policy, Java developers have typically been discouraged from using API features related to scheduling, since the behavior of the method calls can vary so much from one operating system to another. Beginning in Java 1.5, however, features have been added to the Java API to support a more deterministic scheduling policy. The API additions are centered around the use of a thread pool.

Thread pools using Java were covered in Section Section 4.6. In that section, we discussed three different thread-pool structures: (1) a single-threaded pool, (2) a thread pool with a fixed number of threads, and (3) a cached thread pool. A fourth structure, which executes threads after a certain delay or periodically, is also available. This structure is similar to that shown in Figure 4.7. It differs, however, in that we use the factory method `newScheduledThreadPool()` in the `Executors` class, which returns a `ScheduledExecutorService` object. Using this object, we invoke one of four possible methods to schedule a `Runnable` task either after a fixed delay, at a fixed (periodic) rate, or periodically with an initial delay. The program shown in Figure 6.4 creates a `Runnable` task that checks once per second to see if there are entries written to a log. If so, the entries are written to a database. The program uses a scheduled thread pool of size 1 (we only require one thread) and then schedules the task using the `scheduleAtFixedRate()` method so that the task begins running immediately (delay of 0) and then runs once per second.

Exercises

6.1 As discussed in Section Section 6.1, the specification for the JVM may allow implementations to ignore calls to `setPriority()`. An argument in favor of ignoring `setPriority()` is that modifying the priority of a Java thread has little effect once the thread begins running on a native operating-system thread, since the operating-system scheduler modifies the priority of the kernel thread to which the Java thread is mapped based on how CPU- or I/O-intensive the thread is. Discuss the pros and cons of this argument.

```
import java.util.concurrent.*;

class Task implements Runnable
{
  public void run() {
    /**
     * check if there are any entries that
     * need to be written from a log to
     * a database.
     */
    System.out.println("Checking for log entries ...");
  }
}

public class SPExample
{
  public static void main(String[] args) {
    // create the scheduled thread pool
    ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(1);

    // create the task
    Runnable task = new Task();

    // schedule the task so that it runs once every second
    scheduler.scheduleAtFixedRate(task,0,1,TimeUnit.SECONDS);
  }
}
```

Figure 6.4 Creating a scheduled thread pool in Java.

Programming Problems

6.2 In programming problem Exercise 4.7, you wrote a program that listed each thread in the JVM. Modify this program so that you also list the priority of each thread.