Thread Scheduling

As mentioned briefly in the previous section, many computer configurations have a single CPU. Hence, threads run one at a time in such a way as to provide an illusion of concurrency. Execution of multiple threads on a single CPU in some order is called *scheduling*. The Java runtime environment supports a very simple, deterministic scheduling algorithm called *fixed-priority scheduling*. This algorithm schedules threads on the basis of their priority relative to other Runnable threads.

When a thread is created, it inherits its priority from the thread that created it. You also can modify a thread's priority at any time after its creation by using the `setPriority` method. Thread priorities are integers ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority. Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run. The chosen thread runs until one of the following conditions is true:

- A higher priority thread becomes runnable.
- It yields, or its `run` method exits.
- On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

The Java runtime system's thread scheduling algorithm is also preemptive. If at any time a thread with a higher priority than all other `Runnable` threads becomes `Runnable`, the runtime system chooses the new higher-priority thread for execution. The new thread is said to *preempt* the other threads.

Rule of thumb: At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use thread priority only to affect scheduling policy for efficiency purposes. Do not rely on it for algorithm correctness.

A Thread Race

`RaceApplet` is an applet that animates a race between two "runner" threads of different priorities. Clicking the mouse on the applet starts the two runners. Runner 2 has a priority of 2; runner 3 has a priority of 3.

Try this: Click the applet below to start the race.

Note: If you don't see the applet running above, you need to install Java Plug-in, which happens automatically when you install the J2SE JRE or SDK. This applet requires

version 5.0 or later. You can find more information in the Java Plug-in home page.

The runners are implemented by a `Thread` subclass called `Runner`. Here is the `run` method for the `Runner` class, which simply counts from 1 to 10,000,000:

```
public int tick = 1;
public void run() {
    while (tick < 10000000) {
        tick++;
    }
}
```

This applet has a third thread, which handles the drawing. The drawing thread's `run` method loops until the applet stops. During each iteration of the loop, the thread draws a line for each runner, whose length is computed from the runner's `tick` variable; the thread then sleeps for 10 milliseconds. The drawing thread has a thread priority of 4 — higher than that of either runner. Thus, whenever the drawing thread wakes up after 10 milliseconds, it becomes the highest-priority thread, preempting whichever runner is currently running, and draws the lines. You can see the lines inch their way across the page.

This is not a fair race, because one runner has a higher priority than the other. Each time the drawing thread yields the CPU by going to sleep for 10 milliseconds, the scheduler chooses the highest-priority Runnable thread to run; in this case, it's always runner 3.

Here is another applet, one that implements a fair race, in which both runners have the same priority and an equal chance of being chosen to run.

Try this: Click the applet below to start the race.

Note: If you don't see the applet running above, you need to install Java Plug-in, which happens automatically when you install the J2SE JRE or SDK. This applet requires version 5.0 or later. You can find more information in the Java Plug-in home page.

In this race, each time the drawing thread yields the CPU by going to sleep, there are two Runnable threads of equal priority — the runners — waiting for the CPU. The scheduler must choose one of the threads to run. In this case, the scheduler arbitrarily chooses one.

Selfish Threads

The `Runner` class used in the previous races implements impaired thread behavior. Recall the `run` method from the `Runner` class used in the races:

```
public int tick = 1;
public void run() {
    while (tick < 10000000) {
        tick++;
    }
}
```

The `while` loop in the `run` method is in a tight loop. Once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU; it just continues to run until the `while` loop terminates naturally or until the thread is preempted by a higher-priority thread. This thread is called a *selfish thread*.

In some cases, having selfish threads doesn't cause any problems, because a higher-priority thread preempts the selfish one, just as the drawing thread in `RaceApplet` preempts the selfish runners. However, in other cases, threads with CPU-greedy run methods can take over the CPU and cause other threads to wait for a long time, even forever, before getting a chance to run.

Time-Slicing

Some systems limit selfish-thread behavior with a strategy known as time slicing. Time slicing comes into play when multiple Runnable threads of equal priority are the highest-priority threads competing for the CPU. For example, a standalone program, `RaceTest.java`, based on `RaceApplet` creates two equal-priority selfish threads that have this `run` method:

```
public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ",
                               tick = " + tick);
        }
    }
}
```

This run method contains a tight loop that increments the integer `tick`. Every 50,000 ticks prints out the thread's identifier and its `tick` count.

When running this program on a time-sliced system, you will see messages from both threads intermingled, something like this:

```
Thread #1, tick = 50000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 250000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000
```

This output is produced because a *time-sliced system* divides the CPU into time slots and gives each equal-and-highest priority thread a time slot in which to run. The time-sliced system iterates through the equal-and-highest priority threads, allowing each one a bit of time to run, until one or more finishes or until a higher-priority thread preempts them. Note that time slicing makes no guarantees as to how often or in what order threads are scheduled to run.

When running this program on a system that is not time sliced, you will see messages from one thread finish printing before the other thread ever gets a chance to print one message. The output will look something like this:

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
```

```
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

The reason is that a system that is not time sliced chooses one of the equal-and-highest priority threads to run and allows that thread to run until it relinquishes the CPU (by sleeping, yielding, or finishing its job) or until a higher-priority preempts it.

Purity Tip: The Java platform does not implement (and therefore does not guarantee) time slicing. However, some platforms do support time slicing. Your programs should not rely on time slicing, as it may produce different results on different systems.

Relinquishing the CPU

As you can imagine, writing CPU-intensive code can have negative repercussions on other threads running in the same process. In general, try to write well-behaved threads that voluntarily relinquish the CPU periodically and give other threads an opportunity to run.

A thread can voluntarily yield the CPU by calling the `yield` method. The `yield` method gives other threads of the same priority a chance to run. If no equal-priority threads are Runnable, the `yield` is ignored.

Thread Scheduling Summary

- Many computers have only one CPU, so threads must share the CPU with other threads. The execution of multiple threads on a single CPU, in some order, is called scheduling. The Java platform supports a simple, deterministic scheduling algorithm called fixed-priority scheduling.
- Each thread has a numeric priority between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). At any given time, when multiple threads are ready to be executed, the highest-priority thread is chosen for execution. Only when that thread stops or is suspended will a lower-priority thread start executing.
- When all the `Runnable` threads in the system have the same priority, the scheduler arbitrarily chooses one of them to run.
- The Java platform does not directly time slice. However, the system implementation of threads underlying the Thread class may support time slicing. Do not write code that relies on time slicing.
- A given thread may, at any time, give up its right to execute by calling the `yield` method. Threads can yield the CPU only to other threads of the same priority. Attempts to yield to a lower-priority thread are ignored.