

system and many guest operating systems. The host operating system creates and manages the virtual machines, and each virtual machine has a guest operating system installed and applications running within that guest. Each guest operating system may be fine-tuned for specific use cases, applications, and users, including time sharing or even real-time operation.

Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system is at the mercy of the virtualization system as to what CPU resources it actually receives. A given 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system would be even more catastrophic.

The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed that they are scheduling all of those cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

5.6 Operating System Examples

We turn next to a description of the scheduling policies of the Solaris, Windows, and Linux operating systems. It is important to remember that we are describing the scheduling of kernel threads with Solaris and Windows. Recall that Linux does not distinguish between processes and threads; thus, we use the term *task* when discussing the Linux scheduler.

5.6.1 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling where each thread belongs to one of six classes:

1. Time sharing (TS)
2. Interactive (IA)
3. Real time (RT)
4. System (SYS)
5. Fair share (FSS)
6. Fixed priority (FP)

Within each class there are different priorities and different scheduling algorithms.

The default scheduling class for a process is time sharing. The scheduling policy for the time-sharing class dynamically alters priorities and assigns

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 5.12 Solaris dispatch table for time-sharing and interactive threads.

time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices. The higher the priority, the smaller the time slice; the lower the priority, the larger the time slice. Interactive processes typically have a higher priority, CPU-bound processes, a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications—such as those created by the KDE or GNOME window managers—a higher priority for better performance.

Figure 5.12 shows the dispatch table for scheduling time-sharing and interactive threads. These two scheduling classes include 60 priority levels, but for brevity, we display only a handful. The dispatch table shown in Figure 5.12 contains the following fields:

- **Priority.** The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
- **Time quantum.** The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta: the lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).
- **Time quantum expired.** The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered

CPU-intensive. As shown in the table, these threads have their priorities lowered.

- **Return from sleep.** The priority of a thread that is returning from sleeping (such as waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, thus supporting the scheduling policy of providing good response time for interactive processes.

Threads in the real-time class are given the highest priority. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. A real-time process will run before a process in any other class. In general, however, few processes belong to the real-time class.

Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon. Once established, the priority of a system thread does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the system class).

The fixed-priority and fair-share classes were introduced with Solaris 9. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share scheduling class uses CPU **shares** instead of priorities to make scheduling decisions. CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a **project**).

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue. Figure 5.13 illustrates how the six scheduling classes relate to one another and how they map to global priorities. Notice that the kernel maintains 10 threads for servicing interrupts. These threads do not belong to any scheduling class and execute at the highest priority (160–169). As mentioned, Solaris has traditionally used the many-to-many model (Section 4.2.3) but switched to the one-to-one model (Section 4.2.2) beginning with Solaris 9.

5.6.2 Example: Windows Scheduling

Windows schedules threads using a priority-based preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the *dispatcher*. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The **variable class**

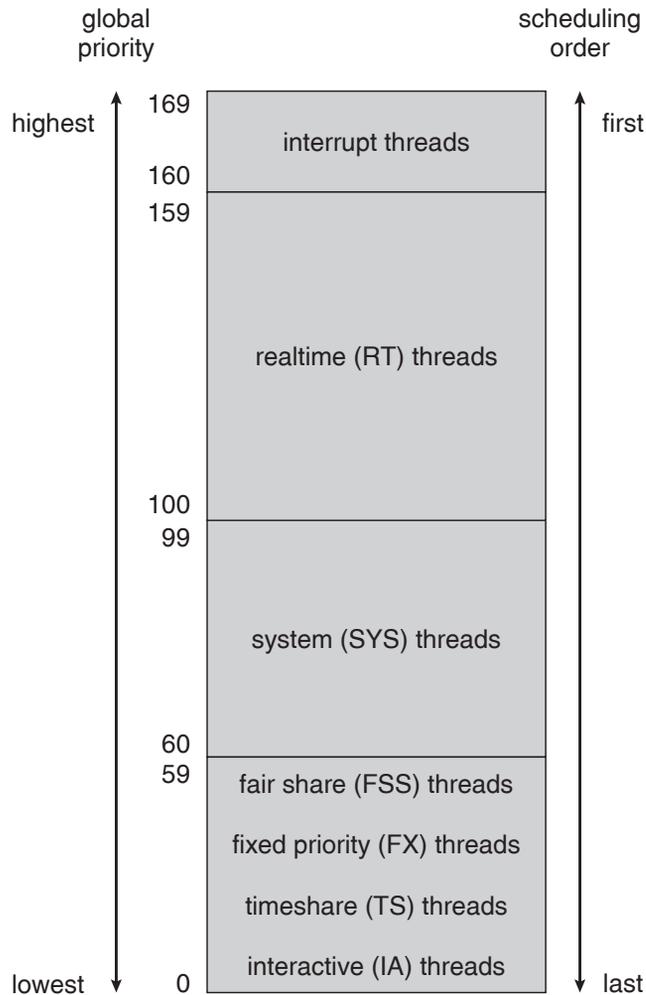


Figure 5.13 Solaris scheduling.

contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows kernel and the Win32 API. The Win32 API identifies several priority classes to which a process can belong. These include:

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 5.14 Windows XP priorities.

- `BELOW_NORMAL_PRIORITY_CLASS`
- `IDLE_PRIORITY_CLASS`

Priorities in all classes except the `REALTIME_PRIORITY_CLASS` are variable, meaning that the priority of a thread belonging to one of these classes can change.

A thread within a given priority classes also has a relative priority. The values for relative priorities include:

- `TIME_CRITICAL`
- `HIGHEST`
- `ABOVE_NORMAL`
- `NORMAL`
- `BELOW_NORMAL`
- `LOWEST`
- `IDLE`

The priority of each thread is based on both the priority class it belongs to and its relative priority within that class. This relationship is shown in Figure 5.14. The values of the priority classes appear in the top row. The left column contains the values for the relative priorities. For example, if the relative priority of a thread in the `ABOVE_NORMAL_PRIORITY_CLASS` is `NORMAL`, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class the thread belongs to. By default, the base priority is the value of the `NORMAL` relative priority for that class. The base priorities for each priority class are:

- `REALTIME_PRIORITY_CLASS`—24
- `HIGH_PRIORITY_CLASS`—13

- ABOVE_NORMAL_PRIORITY_CLASS—10
- NORMAL_PRIORITY_CLASS—8
- BELOW_NORMAL_PRIORITY_CLASS—6
- IDLE_PRIORITY_CLASS—4

Processes are typically members of the NORMAL_PRIORITY_CLASS. A process belongs to this class unless the parent of the process was of the IDLE_PRIORITY_CLASS or unless another class was specified when the process was created. The initial priority of a thread is typically the base priority of the process the thread belongs to.

When a thread's time quantum runs out, that thread is interrupted; if the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for; for example, a thread that was waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several time-sharing operating systems, including UNIX. In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows distinguishes between the *foreground process* that is currently selected on the screen and the *background processes* that are not currently selected. When a process moves into the foreground, Windows increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

5.6.3 Example: Linux Scheduling

Prior to Version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. Two problems with the traditional UNIX scheduler are that it does not provide adequate support for SMP systems and that it does not scale well as the number of tasks on the system grows. With Version 2.5, the scheduler was overhauled, and the kernel now provides a scheduling algorithm that runs in constant time—known as $O(1)$ —regardless of the number of tasks on the system. The new scheduler also provides increased support for SMP, including processor affinity and load balancing, as well as providing fairness and support for interactive tasks.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice** value ranging from 100 to 140. These two ranges map into a global priority scheme wherein numerically lower values indicate higher priorities.

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100			
•			
•			
•			
139			

Figure 5.15 The relationship between priorities and time-slice length.

Unlike schedulers for many other systems, including Solaris (Section 5.6.1) and Windows (Section 5.6.2), Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta. The relationship between priorities and time-slice length is shown in Figure 5.15.

A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta. The kernel maintains a list of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently. Each runqueue contains two priority arrays: **active** and **expired**. The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays contains a list of tasks indexed according to priority (Figure 5.16). The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure. When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged: the expired array becomes the active array, and vice versa.

Linux implements real-time scheduling as defined by POSIX.1b, which is described in Section 5.4.2. Real-time tasks are assigned static priorities. All

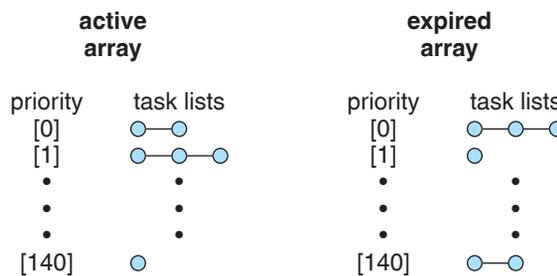


Figure 5.16 List of tasks indexed according to priority.

other tasks have dynamic priorities that are based on their *nice* values plus or minus the value 5. The interactivity of a task determines whether the value 5 will be added to or subtracted from the *nice* value. A task's interactivity is determined by how long it has been sleeping while waiting for I/O. Tasks that are more interactive typically have longer sleep times and therefore are more likely to have adjustments closer to -5 , as the scheduler favors interactive tasks. The result of such adjustments will be higher priorities for these tasks. Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.

A task's dynamic priority is recalculated when the task has exhausted its time quantum and is to be moved to the expired array. Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

5.7 Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? As we saw in Section 5.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 5.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as:

- Maximizing CPU utilization under the constraint that the maximum response time is 1 second
- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

5.7.1 Deterministic Modeling

One major class of evaluation methods is **analytic evaluation**. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

Deterministic modeling is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds: