6. Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The multilevel feedback queue scheduling algorithm, in contrast, allows
a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation. For example, consider a multilevel feedback queue scheduler with three
queues, numbered from 0 to 2 (Figure 6.7).

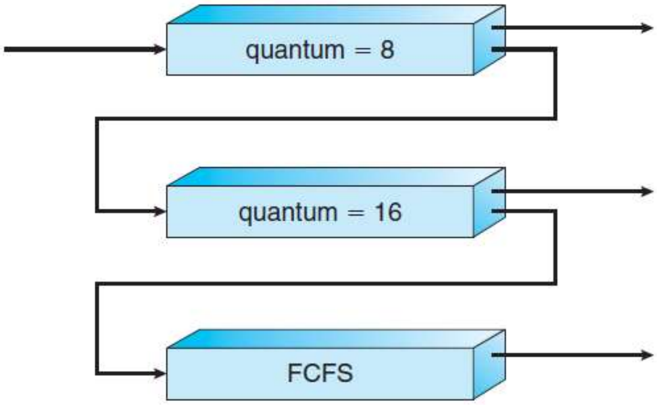


Figure 6.7 Multilevel feedback queues.

The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into

queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

Thread Scheduling

On operating systems that support threads, it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as process contention scope (PCS), since competition for the CPU takes place among threads belonging to the same process.

(When we say the thread library schedules user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses system-contention scope (SCS).

Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

Pthread Scheduling

Pthreads identifies the following contention scope values:

- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD SCOPE PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library. The PTHREAD SCOPE SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC provides two functions for getting—and setting—the contention scope policy:

- pthread attr setscope(pthread attr t *attr, int scope)
- pthread attr getscope(pthread attr t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread attr setscope() function is passed either the PTHREAD SCOPE SYSTEM or the PTHREAD SCOPE PROCESS value, indicating how the contention scope is to be set. In the case of pthread attr getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, load sharing becomes possible—but scheduling problems become correspondingly more complex. Many possibilities have been tried; and as we saw with single processor CPU scheduling, there is no one best solution.

Here, we discuss several concerns in multiprocessor scheduling. We
concentrate on systems in which the processors are identical—homogeneous —in terms of their functionality. We can then use any available processor to run any process in the queue. Note, however, that even with homogeneous multiprocessors, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.

Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code. This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.

Processor Affinity

Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? 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.

Our criteria may include several measures, such as these:

- 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.

1. Deterministic Modeling

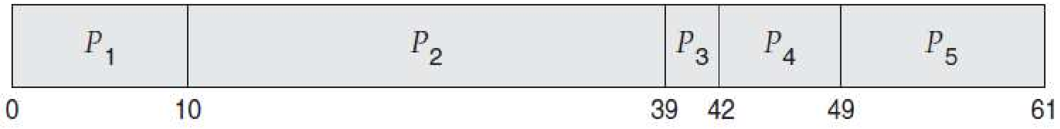
One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload.

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:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

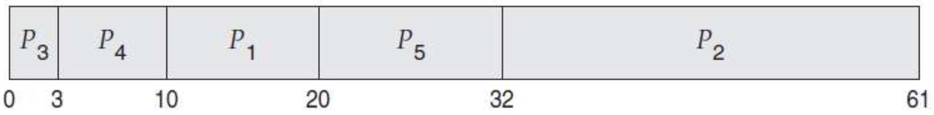
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



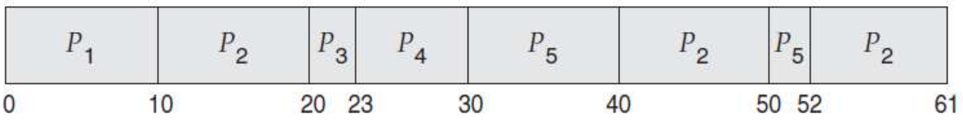
The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as:



The waiting time is 10 milliseconds for process P1, 32 milliseconds for process P2, 0 milliseconds for process P3, 3 milliseconds for process P4, and 20 milliseconds for process P5. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as



The waiting time is 0 milliseconds for process *P*1, 32 milliseconds for process *P*2, 20 milliseconds for process *P*3, 23 milliseconds for process *P*4, and 40 milliseconds for process *P*5. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We can see that, in this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

2. Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst.

Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

3. Simulations

To get a more accurate evaluation of scheduling algorithms, we can use simulations. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system.

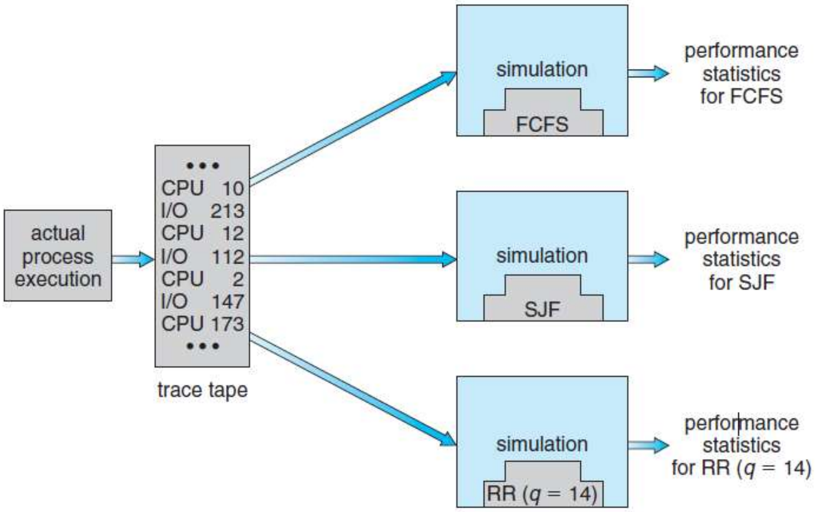


Figure 6.25 Evaluation of CPU schedulers by simulation.

4. Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

The major difficulty with this approach is the high cost. The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures) but also in the reaction of the users to a constantly changing operating system.

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over non interactive processes, then users may switch to interactive use.