CPU Scheduling

In multiprogramming operating systems CPU is switched among processes to improve CPU utilization. In this lecture, we will discuss several CPU-scheduling policies and selection criterion for a particular computer system. The aim of CPU scheduling is assigning CPU to processes to be executed over time, in a way that meets system objectives.

In many systems, this scheduling activity is broken down into three separate functions long/medium/short-term scheduling on relative time scales with which these functions are performed. *Long-term scheduling* is performed when a new process is created. This scheduling function is invoked by the OS, to decide whether to add a new process to the set of processes that are currently active. Medium-term scheduling is a part of the swapping function. This function is invoked by the OS, to decide whether to add a process to those that are at least partially in main memory. Short-term scheduling is the actual decision of which ready process to execute next. Figure-1 illustrates when scheduling functions are invoked in using queues and different states of the process model.

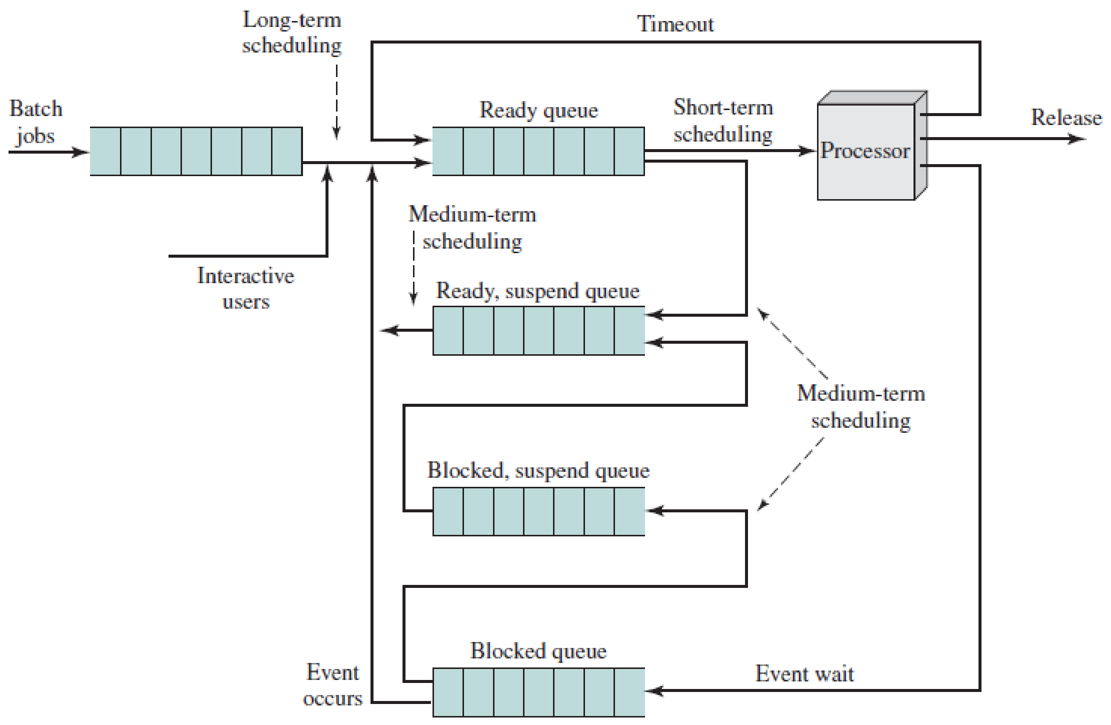


Figure-1: Short/Medium/Long term Scheduling

Long-Term Scheduling determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming. Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler.
Medium-Term Scheduling is part of the swapping function. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming.
Short-Term Scheduling determines to which process CPU is allocated and for how long it will be in ready state.
In terms of frequency of execution, long-term scheduler executes relatively infrequently, medium-term scheduler executes somewhat more frequently, and short-term scheduler executes most frequently.

Process, a program in action. Normally process execution consists of a cycle of CPU execution and I/O wait. Process execution begins with a CPU burst, followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution. During its life cycle from start to termination state, a process alternates between CPU burst and IO burst as shown in Figure-2.

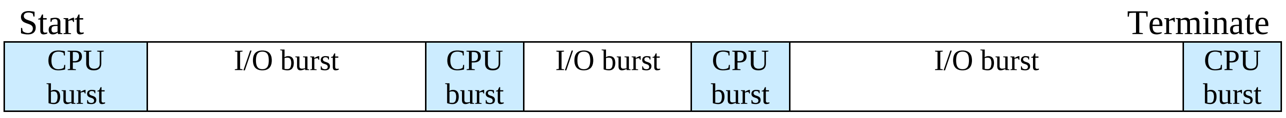


Figure-2.

On the burst basis, a process is defined to CPU/compute bound or I/O bound. If a process spends more time in using CPU during its lifetime (Figure-3), it is called compute bound process. If a process spends more time in performing I/O operations during its lifetime (Figure-4), it is called IO bound process.

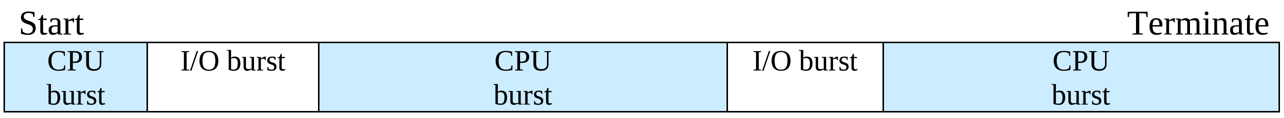


Figure-3: CPU bound process

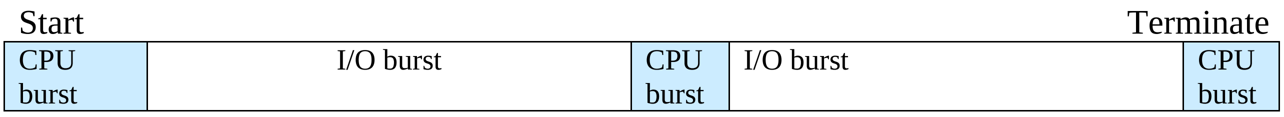


Figure-4: IO bound process

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the selection of a particular algorithm may favor one class of processes over another. In selecting algorithm in a particular situation, we must consider the properties of the various algorithms. Following criteria have been suggested for comparing CPU-scheduling algorithms.

CPU Utilization: To keep the CPU as busy as possible. Conceptually CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent.

Throughput: If the CPU is busy executing processes, then work is being done. The number of processes that are completed per time unit is called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Turnaround Time: The interval from the time of submission of a process to the time of completion is the turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that process. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. The turnaround time is generally limited by the speed of the output device.

Waiting Time: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response Time: In an interactive system, turnaround time may not be the best criterion. Thus, another measure called Response time is the time from the submission of a request until the first response is produced. Response time is the time it takes to start responding, not the time it takes to output the response.

CPU Scheduling Algorithms

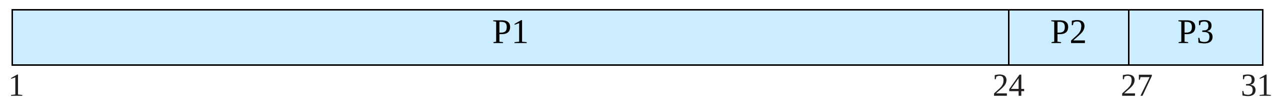
CPU scheduling deals with the decision of allocation of CPU to one of the processes in the ready queue. Now we describe working of different CPU-scheduling algorithms

First-Come, First-Served Scheduling

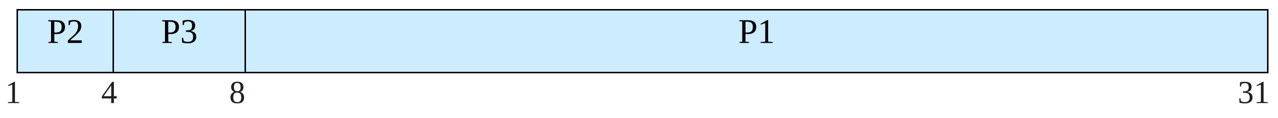
First-Come, First-Served (FCFS) is the simplest CPU-scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The code for FCFS scheduling is simple to write and understand. Average waiting time under the FCFS policy is often quite long. For example, consider the following set of processes that arrive at time 1, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	23
P2	3
P3	4

If the processes arrive in the order P1, P2, P3, in ready queue and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



For process P1 the waiting time is 0 milliseconds, for process *P*2 is 23 milliseconds and for process *P*3 26 milliseconds. Thus, the average waiting time is (0+ 23 + 26)/3 = 17 milliseconds. If the processes arrive in the order *P*2, *P*3, *P*1, in ready queue however, the results will be as shown in the following Gantt chart:



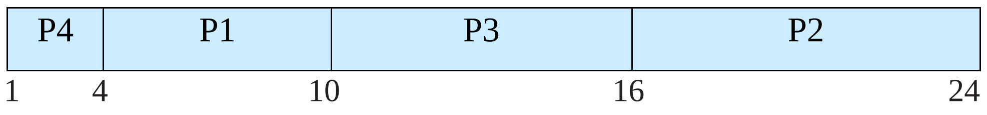
The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds, which is substantial reduction as compared to 17 milliseconds. Thus, the average waiting time under an FCFS policy may vary substantially if the processes' CPU burst times vary greatly. In FCFS approach, there is a convoy effect when all the other processes wait for the one big process to get off the CPU. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first. FCFS scheduling algorithm is non-preemptive. FCFS algorithm is thus particularly no suitable troublesome for time-sharing systems.

Shortest-Job-First Scheduling

This approach associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. Through an example we explain SJF scheduling, for the following set of processes, with the length of the CPU burst given in milliseconds:

.

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



For process P1 waiting time is 3 milliseconds, for process P2 is 16 milliseconds, for process P3 is 9 milliseconds, and for process P4 is 0 milliseconds. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. For the same set of processes, using FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

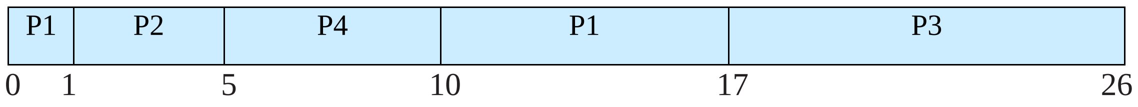
The SJF scheduling algorithm gives the minimum average waiting time for a given set of processes. The real difficulty with the SJF algorithm is knowing the length of the next CPU request. With short-term scheduling, there is no way to know the length of the next CPU burst therefore it cannot be implemented at the level of short-term CPU scheduling. The SJF algorithm is non-preemptive. SJF algorithm with preemptive approach is called *shortest-remaining-time-first* scheduling.

Shortest-Remaining-Time-First (SRTF) Scheduling

In this algorithm when a new process arrives at the ready queue while a previous process is still executing. If the next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, it will preempt the currently executing process.
As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Processes arrival time at the ready queue along with next CPU burst is available, working of SRTF is depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is

$$[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5 \text{ milliseconds.}$$

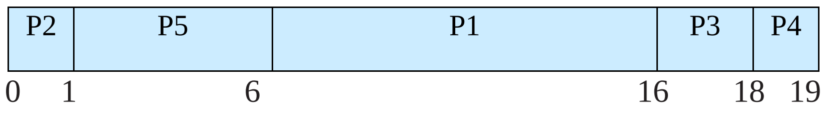
For same set of processes SJF scheduling (non-preemptive) would result in an average waiting time of 7.75 milliseconds

Priority Scheduling

Apriority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. SRTF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Note that we discuss scheduling in terms of high priority and low priority (lecture on OS types, and process model). Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 1,023. Some systems use low numbers to represent low priority; others use low numbers for high priority. Here for further description, we assume that low numbers represent high priority. As an example, consider the following five processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using priority scheduling, these processes are scheduled as depicted in following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either externally or internally. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, and other, often political, factors. Internally defined priorities use some measurable quantity or quantities (time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst) to compute the priority of a process. Normally external priorities are static and internal priorities are dynamic.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is starvation or indefinite blocking. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. A solution to the problem of indefinite wait to get CPU of low-priority processes is aging. Aging approach gradually increase the priority of processes that wait in the system for a long time.

Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

RR scheduling is implemented, by considering ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The RR scheduling algorithm is preemptive. In RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is placed in the ready queue.

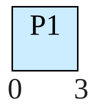
The performance of the RR algorithm depends heavily on the size of the time quantum. If the time quantum is extremely large, the RR policy works like FCFS policy. If the time quantum is small, then short processes will move through the system relatively quickly, i.e. a large number of context switches. One useful guide is that the time quantum should be slightly greater than the time required for a typical interaction or process function.

We can use set of processes used in SRTF scheduling to explore the working of RR scheduling. We use time slice of 3 milliseconds.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	4	5

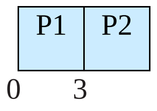
Process P1 is started at time 0, since it is the only process in the queue. It is allocated time slice (0-3). While P1 is in running state, P2, and P3 arrive at time 1 and 2 respectively and are placed in ready queue.

Ready Queue
P2
P3

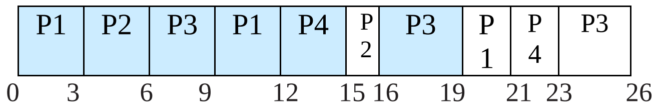


At time 3, P1's time slice expired, so after context it is placed in ready list, and P2 is allocated CPU. While P2 is running (3-6) at time 4, P4 arrives and is placed in ready list as depicted below.

Ready Queue
P3
P1
P4



In this way, processes will be placed in ready list, selected for running state from the front of ready list, and either completing after computation or being context switched to ready. The Gantt chart shows how processes will run to completion in RR scheduling.



Round robin is particularly effective in a general-purpose time-sharing system or transaction processing system. One drawback to round robin is its relative treatment of processor-bound and I/O-bound processes. If there is a mix of processor-bound and I/O-bound processes, then the following will happen: An I/O-bound process uses a processor for a short period and then is blocked for I/O; it waits for the I/O operation to complete and then joins the ready queue. On the other hand, a processor-bound process generally uses a complete time quantum while executing and immediately returns to the ready queue. Thus, processor-bound processes tend to receive an unfair portion of processor time, which results

in poor performance for I/O-bound processes, inefficient use of I/O devices, and an increase in the variance of response time.
To avoid this unfairness, refinement to round robin is suggested, this refined approach is called Virtual Round Robin (VRR).

Virtual Round Robin (VRR) Scheduling
New processes arrive and join the ready queue, which is managed on FCFS basis. When a running process times out, it is returned to the ready queue. When a process is blocked for I/O, it joins an I/O queue. So far, this is as usual and like RR. The new feature is an *auxiliary ready queue* to which processes are moved after being released from an I/O blocked state. When a dispatching decision is to be made, processes in the auxiliary queue get preference over those in the main ready queue. When a process is dispatched from the auxiliary queue, it runs no longer than a time equal to the basic time quantum minus the total time spent running since it was last selected from the main ready queue. This approach is better than round robin in terms of fairness to IO bound processes. Figure-2 depicts Queueing Diagram for Virtual Round-Robin Scheduling.

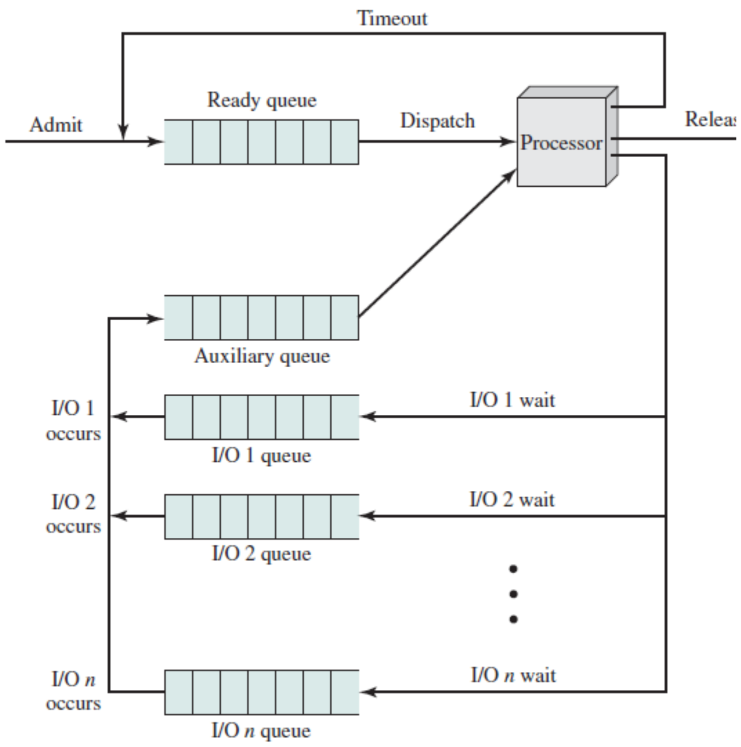


Figure-2: Virtual Round-Robin Scheduling

Multilevel Queue Scheduling
In systems where processes are easily classified into different groups, another class of scheduling algorithms are used. Common division between foreground (interactive) processes and background (batch) processes could be made. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. Foreground queue may have absolute priority over the background queue. By using simple example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

highest priority
1. Interactive processes
2. Interactive editing processes
3. Batch processes
4. Student processes

lowest priority

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on FCFS basis.

Multilevel Feedback Queue Scheduling

A variant of multilevel queue scheduling algorithm, which 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 3). 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.

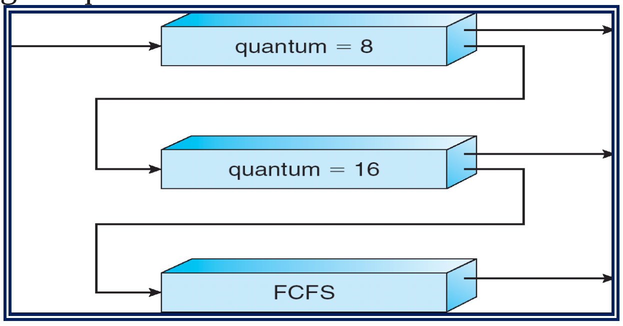


Figure 3: Multilevel feedback queue scheduling

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 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.