# CPU Scheduling

CPU scheduling is the basis of multi-programmed operating system.

By switching the CPU among processes, the operating system can make the computer more productive.

The objective of multiprogramming is to have some process running at all times, in order to maximized CPU utilization.

A process is executed until it must wait, typically for the completion of some I/O request.

In a simple computer system, the CPU then just sits idle; all this waiting time is wasted.

With multiprogramming, we try to use this time productively.

Several processes are kept in memory at one time.

When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.

This pattern continues.

Scheduling of this kind is a fundamental operating-system function.

Almost all computer resources are scheduled before use.

The CPU is one of the primary computer resources.

Thus, its scheduling is central to operating-system design.

## CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes:

Process execution consists of a cycle of CPU execution and I/O wait.

Processes alternate between these two states.

Process execution begins with a CPU burst. That is 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.

The durations of CPU bursts vary greatly from process to process and from computer to computer.

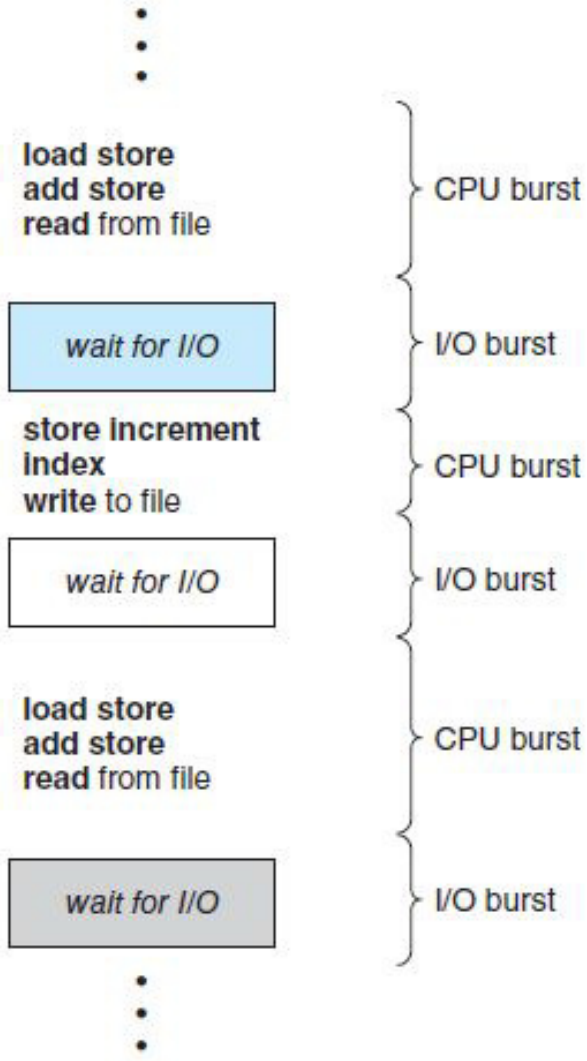



***Fig: Alternating sequence of CPU and I/O bursts.***

An I/O-bound program typically has many short CPU bursts.

A CPU-bound program might have a few long CPU bursts.

This distribution can help us to select an appropriate CPU-scheduling algorithm.

## CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection process is carried out by the short-term scheduler (or CPU scheduler).

The scheduler selects from among the processes in memory, that are ready to execute, and allocates the CPU to one of them.

The ready queue is not necessarily a first-in, first-out (FIFO) queue.

However, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.

The records in the queues are generally process control blocks (PCBs) of the processes.

## Preemptive and Non-preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

i) When a process switches from the running state to the waiting state.

ii) When a process switches from the running state to the ready state (for example, when an interrupt occurs).

iii) When a process switches from the waiting state to the ready state (for example, at completion of I/O).

iv) When a process terminates.

For situations (i) and (iv), there is no choice in terms of scheduling.

A new process (if one exists in the ready queue) must be selected for execution.

There is a choice, however, for situations (ii) and (iii).

When scheduling takes place only under circumstances (i) and (iv), we say that the scheduling scheme is non-preemptive; otherwise, it is preemptive.

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

## Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

This function involves:

i) Switching context from one process to another.

ii) Switching to user mode.

iii) Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, since it is invoked during every context switch.

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

## Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

Many criteria have been suggested for comparing CPU-scheduling algorithms as follows:

**CPU utilization**

We want 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 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

**Throughput**

The number of processes completed per time unit is called throughput.

**Turnaround time**

The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

**Waiting time**

The CPU-scheduling algorithm 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**

The time from the submission of a request until the first response is produced is called response time.

The objective of the CPU scheduling is to maximize CPU utilization and throughput; and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure.

# Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

## First-Come First-Served (FCFS) Scheduling

In FCFS scheduling 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 running process is then removed from the queue.

But the average waiting time under the FCFS policy is often quite long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

If the processes arrive in the order P1, P2, P3; then the Gantt chart for FCFS algorithm is as follows:


The waiting time for process P1, P2 and P3 are 0, 24 and 27 milliseconds respectively.

Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order P2, P3, P1; then the Gantt chart for FCFS algorithm is as follows:


In this case, the waiting time for process P1, P2 and P3 are 6, 0 and 3 milliseconds respectively.

Thus, the average waiting time is (6 + 0 + 3)/3 = 3 milliseconds.

So, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

Another problem may occur if we have one CPU-bound process and many I/O-bound processes.

In this care, 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.

There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU.

This effect results in lower CPU and device utilization.

The FCFS scheduling algorithm is non-preemptive.

Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

## Shortest-Job-First (SJF) Scheduling

The SJF scheduling algorithm associates with the length of the CPU burst of the process.

When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

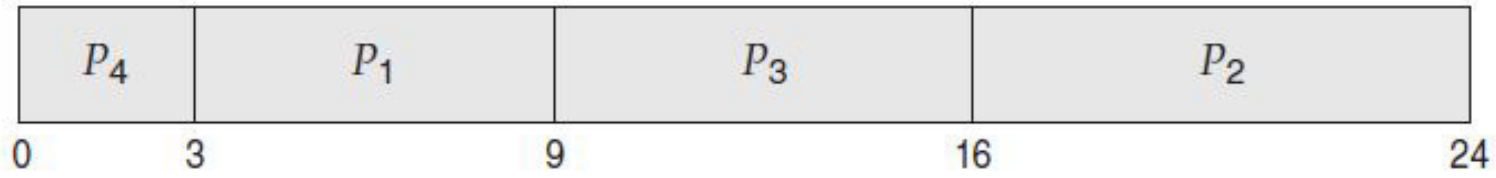
If two processes have the same length next CPU burst, then the FCFS scheduling algorithm is used to break the tie.

The 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.

Consider 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 |

According to the SJF scheduling algorithm, the Gantt chart is given below:




So the waiting time for process P1, P2, P3 and P4 are 3, 16, 9 and 0 milliseconds respectively.

Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

But if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.

By 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.

Therefore, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst.

One approach to this problem is to try to approximate SJF scheduling.

We may not know the length of the next CPU burst, but we may be able to predict its value.

We expect that the next CPU burst will be similar in length to the previous ones.

By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.

We can define the exponential average with the following formula.

Let $t_n$ be the length of the $n^{\text{th}}$ CPU burst, and let $\tau_{n+1}$ be our predicted value for the next CPU burst.

Then, for $\alpha$, $0 \leq \alpha \leq 1$, $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n$.

The value of $t_n$ contains our most recent information, while $\tau_n$ stores the past history.

The parameter $\alpha$ controls the relative weight of recent and past history in our prediction.

If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect.

If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters.

Generally we set $\alpha = 1/2$, so recent history and past history are equally weighted. The initial $\tau_0$ can be defined as a constant or as an overall system average.

We can expand the formula for $\tau_{n+1}$ by substituting for $\tau_n$ to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

Since both $\alpha$ and $(1 - \alpha)$ are less than 1, each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or non-preemptive.

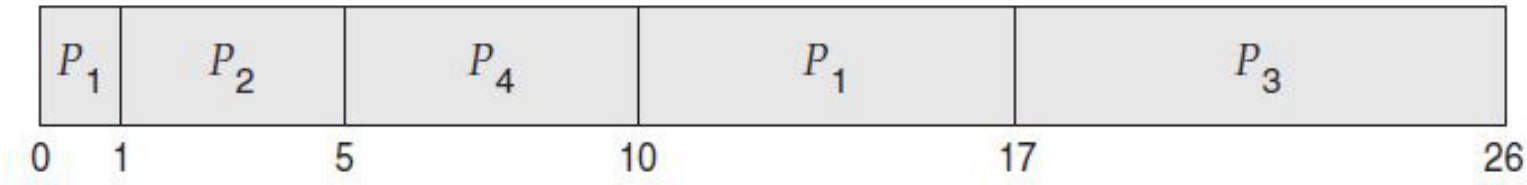
If next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, then a preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called shortest-remaining time-first scheduling.

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 |

In case of Preemptive SJF scheduling algorithm, the Gantt chart is as follows:




The average waiting time is [(10 – 1) + (1 – 1) + (17 – 2) + (5 – 3)]/4 = 26/4 = 6.5 milliseconds.

In case of Non-preemptive SJF scheduling, the average waiting time for this example will be 7.75 milliseconds.

# Priority Scheduling

In priority scheduling algorithm, a priority 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.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.

The larger the CPU burst, the lower the priority, and vice versa.

Priorities are generally indicated by some fixed range of numbers.

Some systems use low numbers to represent low priority; others use low numbers for high priority.

Let consider the following set of processes arrived at the same time in the order P1, P2, P3,P4, P5, and 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 algorithm, the processes are scheduled according to the following Gantt chart:


The average waiting time is (6+0+16+18+1)/5=8.2 milliseconds.

Priorities can be defined either internally or externally.

Internally defined priorities use some measurable quantities like time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst etc.

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, the department sponsoring the work, and other, often political factors. Priority scheduling can be either preemptive or non-preemptive.

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 indefinite blocking, or starvation.

A process that is ready to run but waiting for the CPU can be considered blocked.

A priority scheduling algorithm can leave some low priority processes waiting indefinitely for the CPU.

A solution to the problem of indefinite blockage of low-priority processes is **aging**.

Aging is the method of gradually increasing the priority of processes that waiting the system for a long time.

## Round-Robin Scheduling

The Round-Robin (RR) Scheduling algorithm is designed for time-sharing system.

It is similar to FCFS scheduling, but preemption is added to switch between processes.

A small unit of time, called a time quantum or time slice, is defined.

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.

To implement RR scheduling, 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.

If the process has a CPU burst less than 1 time quantum, then the process itself will release the CPU voluntarily, and 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.

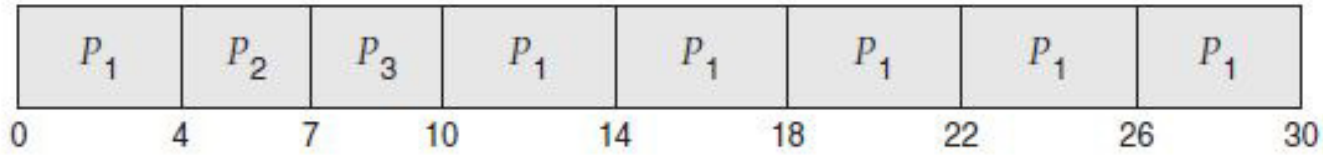
A context switch will be executed, and the process will be put at the tail of the ready queue.

The CPU scheduler will then select the next process in the ready queue.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

If the time quantum is 4 milliseconds, then the Gantt chart is as follows:




The average waiting time is ((10-4)+4+7)/3=5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum.

The RR scheduling algorithm is thus preemptive.

If there are $n$ processes in the ready queue and the time quantum is $q$, then each process gets $1/n$ of the CPU time in chunks of at most $q$ time units.

Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum.

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 is the same as the FCFS policy.

If the time quantum is extremely small, then the RR approach can result in a large number of context switches.

So, the time quantum should be large with respect to the context switch time.

If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.

Turnaround time also depends on the size of the time quantum.

Generally, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

# Multilevel Queue Scheduling

In multilevel queue scheduling algorithm processes are classified into different group.

A common division is made between foreground (interactive) processes and background (batch) processes.

These two types of processes have different response-time requirements.

In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queue.

The processes are permanently assigned into one queue based on some priority of the process like memory size, process priority or process type.

For example, separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm.

The foreground queue might be scheduled by an RR algorithm, whereas the background queue is scheduled by an FCFS algorithm.

There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

Consider the example of a multilevel queue scheduling algorithm with our queues, listed below in order of priority:

1. Real-time processes

2. System processes

3. Interactive processes

4. Batch processes

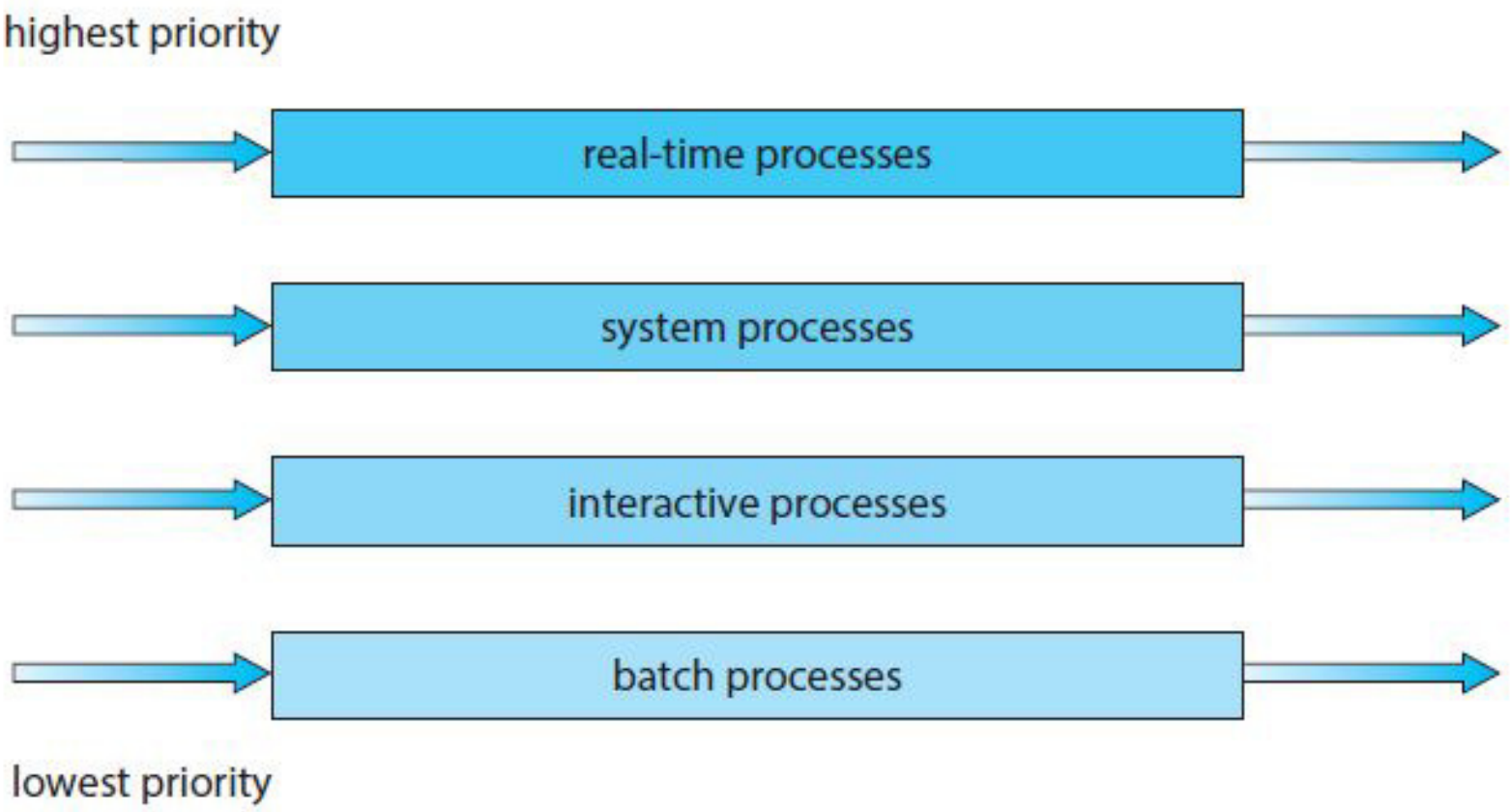



***Fig: Multilevel queue scheduling.***

Each queue has absolute priority over lower-priority queues.

For example, no process in the batch queue could run unless the queues for real-time processes, system processes, and interactive processes were all empty.

If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Additionally, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes in its queue.

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 an FCFS basis.

## Multilevel Feedback Queue Scheduling

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—which are typically characterized by short CPU bursts—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.

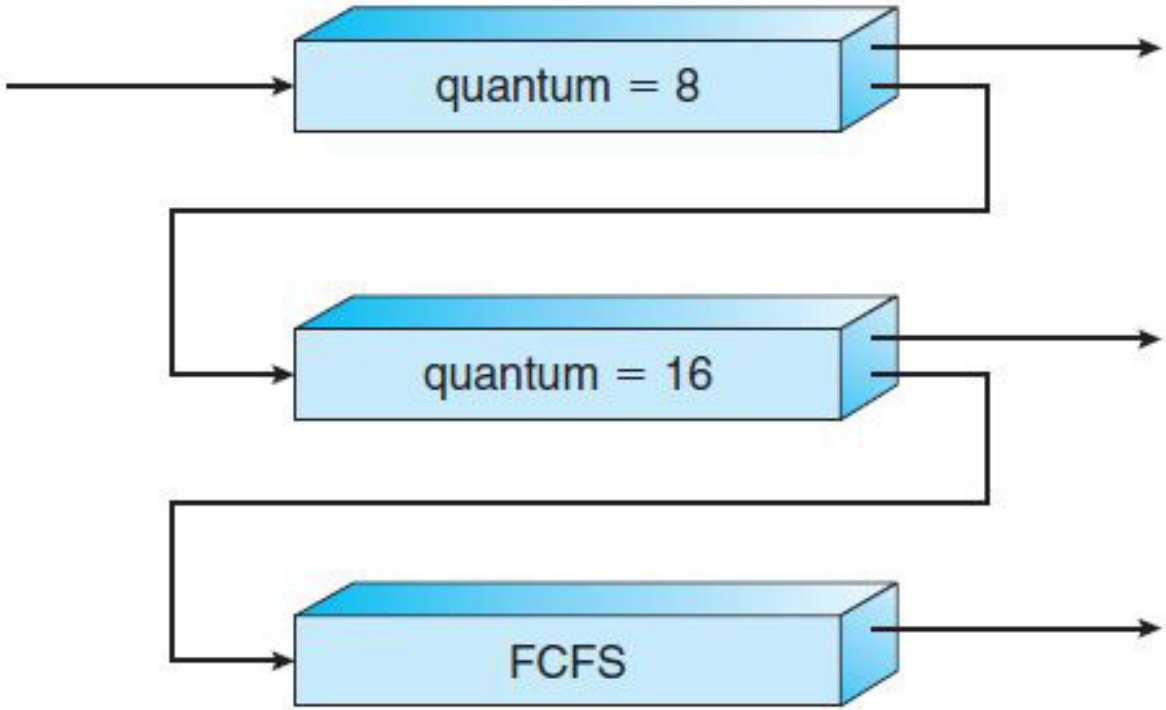



***Fig: Multilevel feedback queues.***

Consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2.

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.

An entering process 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.

Generally a multilevel feedback queue scheduler is defined by the following parameters:

i) The number of queues.

ii) The scheduling algorithm for each queue.

iii) The method used to determine when to upgrade a process to a higher priority queue.

iv) The method used to determine when to demote a process to a lower priority queue.

v) The method used to determine which queue a process will enter when that process needs service.

The multilevel feedback queue scheduler is the most general as well as most complex CPU-scheduling algorithm.

## Algorithm Evaluation

We have to select a CPU Scheduling algorithm for a particular system.

There are many scheduling algorithms, each with its own parameters.

Therefore, selecting an algorithm can be difficult.

To select an algorithm, some criteria should be defined.

The criteria is often defined in terms of CPU utilization, response time, or throughput.

To select an algorithm, we should define the relative importance of these elements.

The criteria may include several measures, such as these:

i) Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds.

ii) 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 various algorithms under consideration.

## 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.

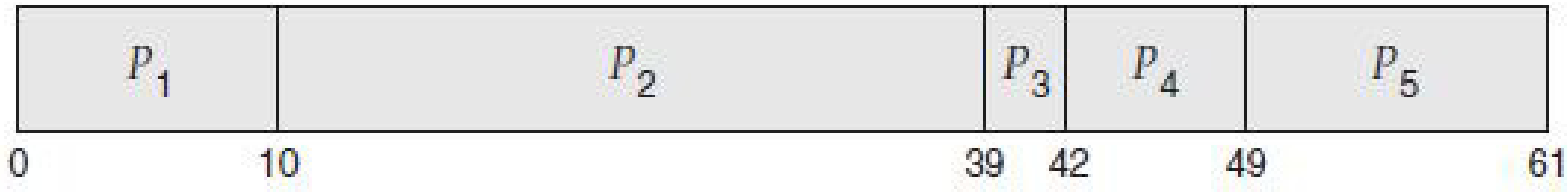
This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Consider the following workload:

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

The arrival time of all 5 processes are 0, and we consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes.

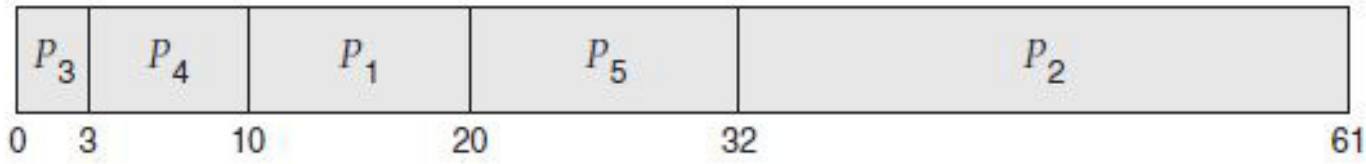
For the FCFS algorithm, the Gantt chart is as follows:




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.

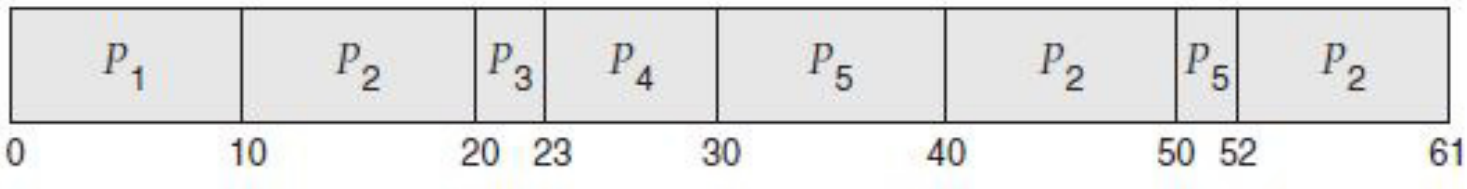
For the SJF algorithm, the Gantt chart is as follows:




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.

For the RR (quantum = 10 milliseconds) algorithm, the Gantt chart is as follows:




The waiting time is 0 milliseconds for process P1, 32 milliseconds for process P2, 20 milliseconds for process P3, 23 milliseconds for process P4, and 40 milliseconds for process P5.

Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

It can be observed 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.

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.

## Queueing Models

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.

However, the distribution of CPU and I/O bursts can be determined.

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).

Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on.

This area of study is called queueing-network analysis.

let $n$ be the average long-term queue length (excluding the process being serviced), $W$ be the average waiting time in the queue, and $\lambda$ be the average arrival rate for new processes in the queue.

We expect that during the time $W$ that a process waits, $\lambda \times W$ new processes will arrive in the queue.

If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W$$

This equation, known as Little's formula.

This formula is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations.

At the moment, the classes of algorithms and distributions that can be handled are fairly limited.

The mathematics of complicated algorithms and distributions can be difficult to work with.

Therefore, although queueing models are often only approximations of real systems, but the accuracy of the computed results may be questionable.