**CPU Scheduling**

In a single core system, only one process can run at a time. Other processes must wait until the core is free and then be rescheduled. In contrast, a multiprogramming system runs several processes at the same time. This is done to maximize CPU utilization.

A process is executed until it has to wait (I/O request). In a simple system, CPU just sits idle, all the waiting time is wasted as no useful work is done. In a multiprogramming system, this time is utilized productively. Several processes are kept in the memory. When a process needs to wait for an event, the OS takes the CPU away from the process and gives to another process. This process is continued for the remainder of the processes. On a multicore system this concept is expanded to all the processing cores of the system. Almost all computer resources are scheduled before use.

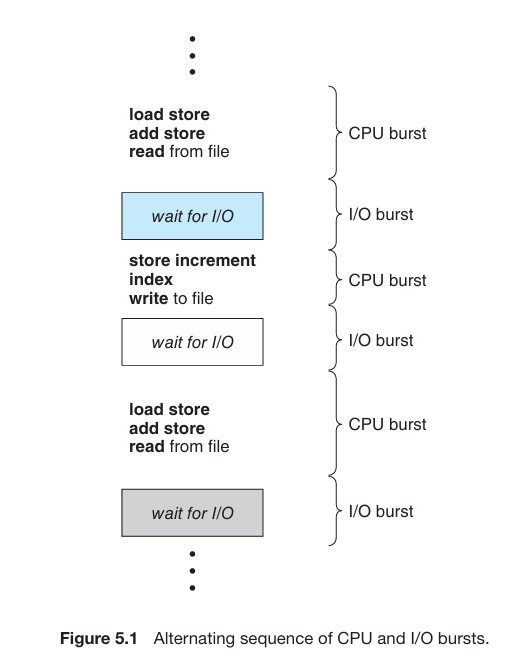
**CPU-I/O Burst Cycles**

Success of CPU scheduling depends on observed property of a process. Process execution consists of a **cycle** of CPU execution and I/O wait; it alternates between these two states. A process starts execution in **CPU burst**. It is followed by **I/O burst** then followed by another CPU burst and then I/O burst and so on. The final CPU burst ends with a system request to terminate the process.

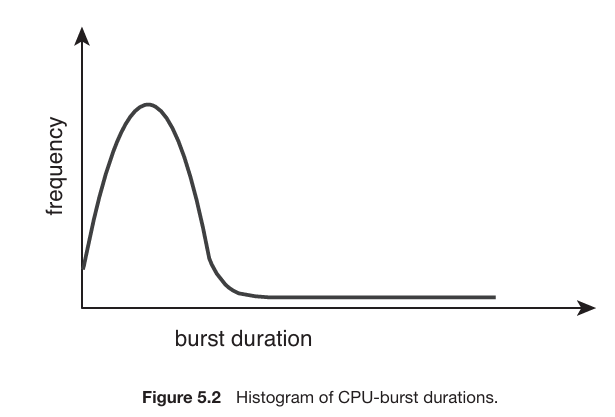
**Cycle** – Unit of time during which CPU executes instructions or performs tasks

**CPU burst** – A period during which a process actively utilizes CPU to execute its instructions is CPU burst.

**I/O burst** – A period during which a process must wait for an I/O operation to complete. It typically happens after CPU burst, during which the process initiates an I/O request.



The CPU bursts vary from process to process and from computer to computer. But they have a similar frequency curve:



The curve is typically described as exponential or hyper exponential. It consists of large number of small CPU bursts and small number of long CPU bursts.

|  |  |
| --- | --- |
| **CPU bound** | **I/O bound** |
| Can have long CPU bursts | Consists of many small CPU bursts |

**CPU Scheduler**

Whenever the CPU is idle, the OS must select a process from the ready queue to be executed. This is done by **CPU scheduler**.

**CPU scheduler** – Selects a process from the ready queue and allocates the CPU to it.

The ready queue is not necessarily a FIFO (First in First Out) queue. It can be implemented as a priority queue, a normal queue, a tree or a linked list. It depends on the scheduling algorithm. All process lined up in the ready queue are waiting for a chance to run on the CPU. The records/nodes in the queue are generally PCBs (Process Control Block) of the processes.

**Preemptive and non-preemptive scheduling**

CPU scheduling decisions may take place under the following circumstances:

1. The process terminates
2. The process goes from running to waiting state. It could occur due to I/O request or invoking of wait () for child termination.
3. Process goes from running to ready state. This occurs because of an interrupt.
4. Process goes from waiting to ready state. This happens after completion of an I/O event.

For situations 1 and 2 there is no scheduling choice. A new process must be selected from the ready queue for execution. The situations 1 and 2 are **non-preemptive** scheduling schemes (also known as *cooperative*). Otherwise it is **preemptive**.

|  |  |
| --- | --- |
| **Non-Preemptive** | **Preemptive** |
| Once the process has been allocated to the CPU, it occupies it until it is terminated or it switches to a waiting state. | The OS can interrupt a running process and can assign the CPU to another process based on priority or pre-defined scheduling algorithm. |

**Difference Table**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Preemptive** | **Non-Preemptive** |
| Interruption | The OS can interrupt a running process to allocate CPU to another process | The OS cannot interrupt the running process until it releases the CPU voluntarily. |
| Context switching | Requires frequent context switching among processes | Context switching takes place only after a process releases the CPU |
| System responsiveness | Better system responsiveness as priority/critical processes can be executed promptly | Can lead to poor system responsiveness if a long-running process takes hold of CPU |
| Implementation Complexity | More complex due to frequent interrupts handling and scheduling decisions. | Simpler. Does not involve frequent interrupt handling and scheduling decisions. |
| Race Condition | May lead to race condition during context switching when accessing shared resources concurrently. | Less prone to race condition as process releases CPU after completion or voluntarily, reducing concurrent access to shared resources. |
| Examples | Round-Robin, Priority scheduling, Shortest Remaining Time First (SRTF) | First Come, First Served (FCFS), Shortest Job Next (SJN) |

Preemptive and non-preemptive scheduling affects kernel design as well.

**Non-Preemptive kernel** – It waits for a system call to complete or for a process to block while waiting for I/O to complete before switching to another process (context switch)

**Preemptive kernel** – A type of OS kernel which supports preemptive scheduling is a preemptive kernel. It can preempt the execution of a process even when it is inside a system call.

The sections of code affected by interrupts must be guarded by OS from simultaneous use. The kernel disables interrupts upon entering the said region of code and re-enables them upon exit. This prevents concurrent access to the code section. These sections of code do not occur typically and consists of very few instructions.

**Dispatcher**

Another component involved in the CPU scheduling function is the **dispatcher**.

**Dispatcher –** A module which gives control of the CPU’s core to the process selected by the CPU scheduler.

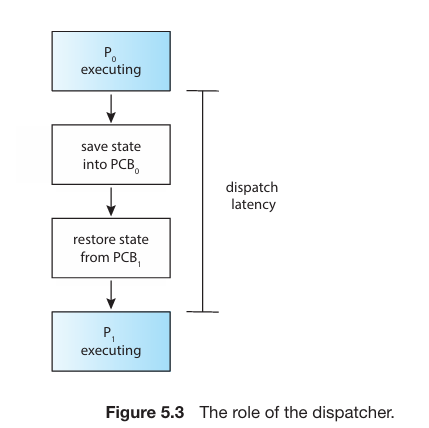
The dispatcher function involves the following:

* Switch context from one process to another
* Switching to user mode
* Jumping to the proper location in the user program to resume that program

The dispatcher must be as fast as possible as it is invoked during every context switch. This is done to:

* **Minimize Overhead** – A fast dispatcher ensures less time is wasted during context switching and more is spent on executing processes.
* **Maintaining responsiveness** – A fast dispatcher ensures better response of the system. The system can respond to user requests more frequently and efficiently
* **Efficient resource allocation** – A fast dispatcher enables better OS response time to changes in process priorities, resource availability, and system overload allowing it to make better scheduling decisions.

**Dispatch latency –** Refers to the time taken by the dispatcher to stop one process and start another process.



|  |  |
| --- | --- |
| **Voluntary** | **Non-Voluntary** |
| Occurs when a process gives up control of the CPU because it requires resource which is currently unavailable (blocking for I/O). | Occurs when the CPU has been taken away from a process (time slice expiring or preempted by a higher-priority process). |

**Scheduling Criteria**

Each scheduling algorithm has different properties and the choice of an algorithm depends on the class of processes. Basically, in selecting an algorithm its properties must be taken into account.

Many criteria have been suggested for comparing different scheduling algorithms. The characteristics to be used for comparison may affect the choice of algorithm.

|  |  |
| --- | --- |
| **Criteria** | **Description** |
| **CPU Utilization** | We want to keep the CPU as busy as possible. Utilization could range from 0 to 100 percent. In real world it is typically 40% (lightly loaded systems) and 90% (heavily loaded systems). |
| **Throughput** | It is referred to as the number of processes completed per unit time. Typically called work done by a CPU. For long process it may be one process over several seconds, for shorter it may be tens of processes per second. |
| **Turnaround Time** | The interval of time of submission of a process and the completion of the process is turnaround time. It is the sum of periods spent waiting in the ready queue, executing on the CPU and doing I/O. |
| **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 interactive systems, turnaround time may not be the best criterion. One process may generate result fairly early and continue computing while the results are being displayed. Hence, another measure is the time of request submission until generation of first response. This is called response time. It is the time taken to start responding, not displaying output of the response. |

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. But in special cases, we optimize minimum and maximum values/measures rather than average. For example, if we want all users to get good service, we will minimize the maximum response time.

In an interactive system, it is more important to minimize the variance in response time rather than the average response time. A system with reasonable and predictable response time may be considered more desirable than a system which is faster on average but higher in variance.

**Scheduling Algorithms**

CPU scheduling deals with the problem of deciding which process in the ready queue is to be allocated to the CPU’s core.

There are many different types of CPU scheduling algorithms. Although modern CPU architectures have multiple processing cores, hence these algorithms are described in context of a single processor system.

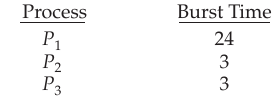
**First come, First Served (FCFS) Scheduling**

It is by far the simplest scheduling algorithm. The FCFS policy is maintained by a FIFO queue.

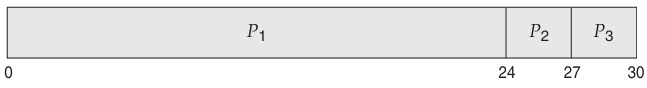
**FCFS** – The process which requests the CPU first, is allocated the CPU first.

When a process enters the ready queue, its PCB is linked to the tail of the queue. When the CPU is free, the process at the head of the queue is allocated to the CPU. The running process is then removed from the queue.

On the negative side, the waiting time is often too long. Consider the following example of processes with arrival time 0 and CPU burst time in milliseconds:



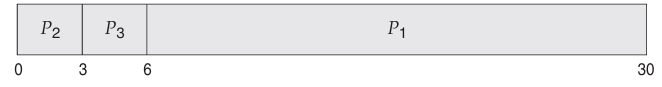
If the processes were to arrive in the order illustrated above (arrival time 0), we will get the result shown in the **gnat chart** below:

****

**Gnat chart** – It is a bar chart which illustrates a particular schedule, including the start and finish times of the participating processes.

The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in reverse order, then the gnat chart will look like:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes’ CPU burst times vary greatly.

Additionally, if we consider a dynamic situation where there are multiple I/O bound and CPU bound processes, FCFS scheme may result in low CPU and device utilization. The CPU bound process takes a hold of the CPU during which the I/O dependent process will complete their I/O and move into ready queue. While the I/O processes wait in the queue, devices are idle. When the CPU bound process finishes its CPU burst it will move on to I/O device. Meanwhile all other I/O bound processes which have small CPU burst time will execute quickly and move to I/O queue. At this point the CPU sits idle. This process is repeated.

The above example describes *convoy effect* as all other processes wait for one big process to relinquish CPU. This results in lower CPU and device utilization.

**Convoy effect** – It refers to a phenomenon where multiple processes in the ready queue are scheduled together by the OS onto a single processor core.

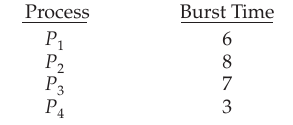
FCFS is a *non-preemptive*scheduling algorithm thus; it is particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals.

**Shortest Job First (SJF) Scheduling**

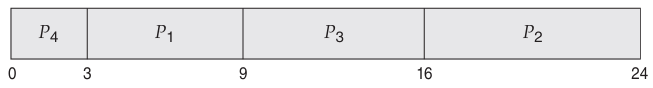
This algorithm associates with each process the length of process’s next CPU burst.

When the CPU is available, it is assigned the process with smallest next CPU burst. If the bursts are same for the two processes, FCFS is used to break the tie.

Basically the algorithm depends on the shortest next CPU burst of the process. Consider the following examples of processes with bursts in milliseconds:



The following will be the resulting gnat chart using SJF:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.Thus,the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

SJF is provably optimal as it gives the minimum average waiting time for a given set of processes. Moving a short process before a long processes decreases waiting time of short process more than it increases the waiting time of the long process. Hence the average waiting time decreases.

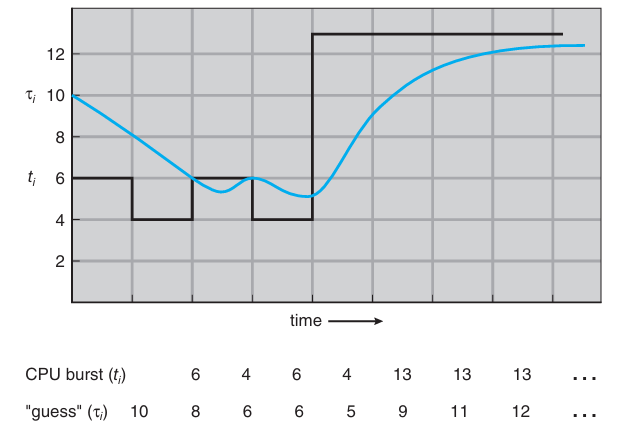
Despite it being optimal, SJF cannot be implemented at CPU scheduling level as there is no way to know the length of the next CPU burst. One approach to this is approximating SJF. It can be done by predicting the value of the next CPU bursts by looking at the previous ones (assuming the next one to be similar). By computing the approximating length of next CPU burst, we can pick the process with the shortest one.

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

Let tn be the length of the nth CPU burst, and let τn+1 be our predicted value for the next CPU burst. Then α can be set in range 0 ≤ α ≤ 1.



The value of tn contains our most recent information, while τn stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If α =0, then τn+1 = τn, and recent history has no effect (current conditions are assumed to be transient). If α =1, then τn+1 = tn and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, α = 1/2, so recent history and past history are equally weighted. The initial τ0 can be defined as a constant or as an overall system average.



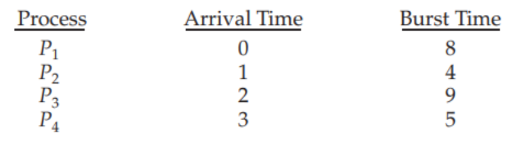
The above figure illustrates exponential average with α =1/2 and τ0 = 10.

SJF can be preemptive and non-preemptive. Choice arises when a new process arrives in the ready queue while a previous one is being executed. The preemptive or non-preemptive quality is described through the CPU burst of the new process.

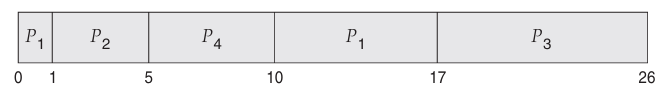
A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

**Shortest Remaining Time First (SRTF) Scheduling**

Preemptive SJF is also known as shortest remaining time first. Consider the following example (time is in milliseconds):



If the processes arrive at the time shown in the arrival time column the resulting gnat chart will look like:



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 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

**Round Robin (RR) Scheduling**

It is similar to FCFS scheduling but preemption is added to allow switching between processes.

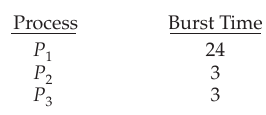
A small slice of time called **time slice** or **time quantum** is defined. It is usually from 10 to 100 milliseconds in length. The ready queue is treated as circular queue; the scheduler traverses the queue allocating CPU core to each process for a time interval up to 1 time quantum.

**Time slice/quantum** – It refers to the maximum amount of time a process is allowed to run in a preemptive multitasking environment before it is interrupted and switched out in favor of another process.

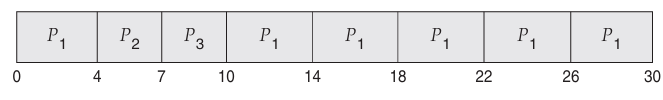
The CPU scheduler picks the first process from the ready queue (FIFO), sets a timer to interrupt after one quantum and dispatches the process. After this one of two things will happen:

1. Process has a CPU burst less than 1 quantum. In this case it will relinquish the CPU voluntarily. The scheduler will proceed to the next process in the queue.
2. Process has a CPU burst greater than 1 quantum. In this case timer will go off causing an interrupt to the OS. A context switch will take place and the current process will be placed in the ready queue and the scheduler will proceed to the next process in the queue.

Consider the following example:



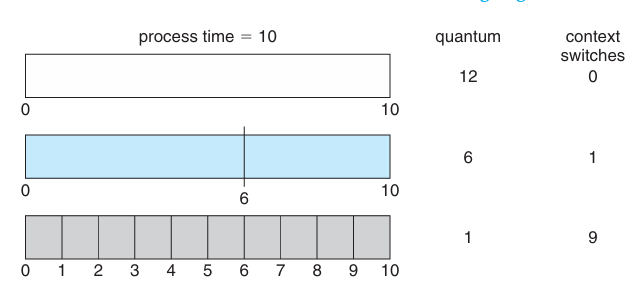
The resulting gnat chart will look like:



If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3.Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. P1 waits for 6 milliseconds (10 − 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, *the average waiting time* is 17/3 = 5.66 milliseconds.

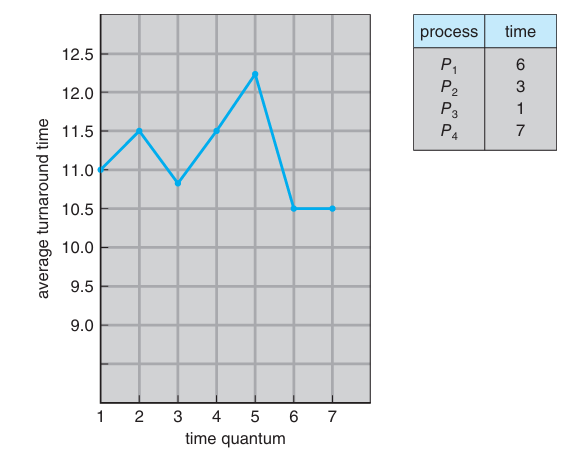
If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n−1) × q time units until its next time quantum.

The performance is heavily dependent on the time quantum. Time quantum is inversely proportional to context switch. If the quantum is large few number of switches will occur (can be 0 if quantum is too large), contrastingly if the quantum is small large number of switches will occur.



Thus we want the quantum time to be as large as possible. If the context-switch time is n percent of the quantum time (approximately), then about n percent CPU time is spent on context switching. Thus the context switch time is a fraction of time quantum.

Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.



Turnaround time also depends on the size of the time quantum. As we can see from the above figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

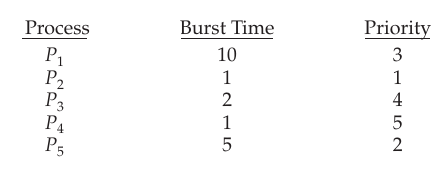
**Priority Scheduling**

In this algorithm priority is associated with the process and CPU is allocated to one whose priority is the highest. SJF is a special case of the general purpose priority scheduling algorithm.

An SJF 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.

Scheduling here is discussed through *high* and *low* priority. Priorities are generally indicated by a range of numbers from 0 to 7 or from 0 to 4,095. Some systems use 0 to represent the highest priority while others may use 0 to denote the lowest.

In the example, we will represent high priority with low numbers. Consider the set of processes at arrival time 0 with CPU bursts in milliseconds:



The resultant gnat chart would look like:



|  |  |
| --- | --- |
| **Internal** | **External** |
| Internally defined priority uses some measureable quantity or quantities for computing priority. For example, time limits, memory requirement and average ratio of I/O bursts to CPU bursts. | Externally defined priority is set by criteria outside the operating system. For example, importance of process, funds being paid for system use and often political factors. |

Priority scheduling can be preemptive and non-preemptive as well. In a preemptive algorithm the priority of incoming process is compared with the running one. If higher, then the process is preempted and the higher priority one starts execution. However, in a non-preemptive algorithm, higher priority process will be placed at the head of the queue.

A major problem with priority scheduling is **indefinite blocking** or **starvation**.

**Indefinite blocking** – It refers to a situation where one or more processes are unable to proceed as they are waiting for resources held by other processes which are dependent on other resources held by another set of processes.

**Starvation** – It refers to a situation where a process is perpetually denied access to necessary resources (CPU time or I/O access) and is unable to make progress towards its completion despite being in ready to execute.

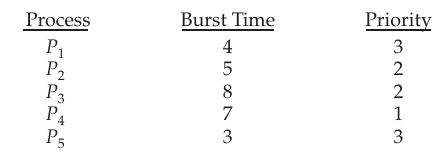
In a heavily loaded system, low level priority process can be left waiting indefinitely. A stream of high priority processes could prevent low priority processes to ever get the CPU. Either of the two will happen; the processes will run or the system would crash and will lose all unfinished low priority processes.

A solution to the above mention problem(s) is **aging**

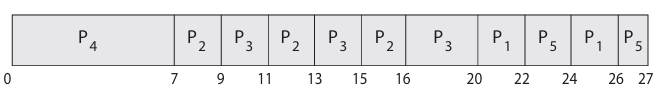
**Aging** – It refers to the increase in priority of processes that wait in the system for a long time from low to high over a period of time.

Eventually every process with a low priority will have a high one and will be executed.

Another solution will be to use RR scheduling which will schedule same priority process using RR scheme while high level ones are executed via priority. Consider the following example with burst time in millisecond and quantum time 2 milliseconds:



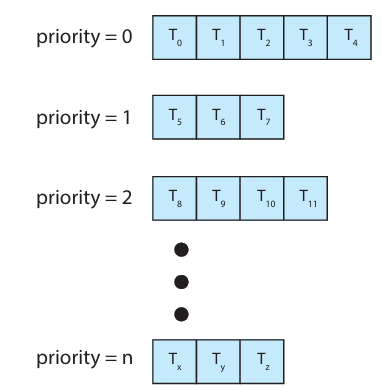
Resulting gnat chart will look like:



**Multilevel Queue Scheduling**

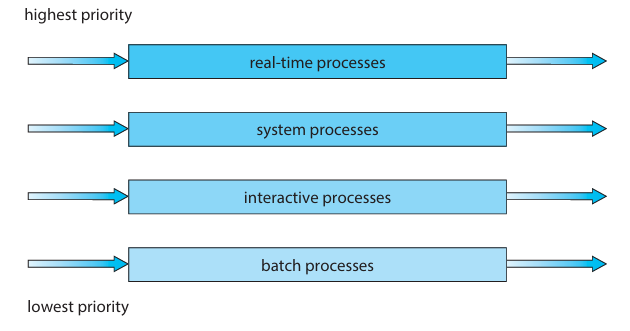
In this approach, each priority has its own separate queue. In priority and round robin scheduling all the processes are placed in a single queue and then the one with the highest priority is selected by the scheduler. Searching a process will require O (n) time hence a multilevel queue is utilized.

A priority is assigned to each process and it remains in that queue for the remainder of its time. If there are multiple processes in queues then they are executed by round robin order.



Above figure illustrates multilevel queue scheduling where each priority has its own separate queue.

A multilevel queue can also be used to partition processes based on their types. A common division is made between **foreground** and **background** processes. These two types of processes have different response time requirements and scheduling needs. Foreground processes may have priority (externally defined) over background processes. Hence separate queues can be used for foreground and background processes. Each may have its own scheduling algorithm.



The above shows the order of priority of each queue. The scheduling is done via fixed priority preemptive scheduling scheme. Each queue has absolute priority over lower-priority queue. No process in the batch queue, for example, 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.

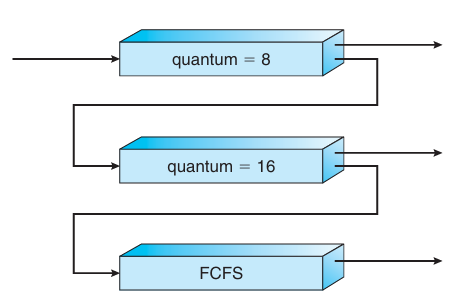
Another possibility is to assign time-slice to each queue. Here each queue will get CPU for a certain period of time which it can then schedule amongst its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR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

**Multilevel Feedback Queue Scheduling**

The multilevel queue scheduling had an advantage of minimum scheduling overhead as a process upon entering the system is assigned to a queue permanently. However this approach is inflexible.

Multilevel feedback queue scheduling in contrast allows processes to move between queues. Processes are separated 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. Additionally, if a process waits for a long period of time in low priority queue it is moved to high priority. This is another form of aging which prevents starvation.

For example, 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. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

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.

The multilevel feedback queue scheduler is defined over the following parameters:

* Number of queues
* Scheduling algorithm for each queue
* The method used to determine when to upgrade process to higher priority queue.
* The method used to determine when to demote a process to lower priority queue.
* The method used to determine which queue a process will enter when it requires service.