time on the ready queue. It has the drawback that the average waiting time is often quite long because FCFS with preemptive time slicing is known as round robin. It does not give good CPU utilization due to the convoy effect. [1][2][14]

We Assume four processes arriving at time = 0, with increasing burst time (P1 = 14, P2 =34, P3 = 45, P4 = 62) as shown in Table2.1. Figure2.1 shows Gantt chart for FCFS.

Table 2-1 FCFS Scheduling

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	14
P2	0	34
P3	0	45
P4	0	62

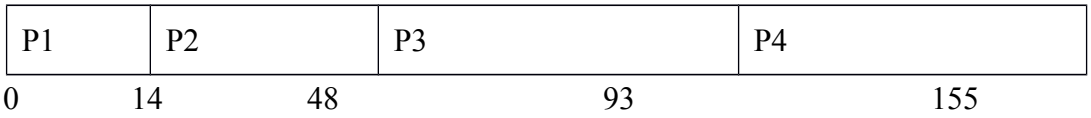


Figure 2-1 Gantt chart for FCFS

Number of Context Switches = 03

Waiting time of P1 = 0

Waiting time of P2 = 14

Waiting time of P3 = 48

Waiting time of P4 = 93

Average Waiting Time = [(P1+P2+P3+P4)]/4

= (0+14+48+93)/4

= 155/4

= 38.75 ms

Turnaround Time for P1 = 14

Turnaround Time for P2 = 48

Turnaround Time for P3 = 93

Turnaround Time for P4 = 155

Average Turnaround Time = [(P1+P2+P3+P4)]/4

= (14+48+93+155)/4

= 310/4

= 77.5 ms

We Suppose four processes arriving at time = 0, with increasing burst time (P1 = 25, P2 =5, P3 = 10, P4 =5) as shown in Table 2-2.

Table 2-2 FCSS Performance

Pid	**Reqd. time**	**Waiting time**	**Turnaround Time**	**Penalty Ratio = 1/Response ratio**
P1	25	0	25	1
P2	5	25	30	6
P3	10	30	40	4
P4	5	40	45	9
Average		23.75		5

Throughput = 4/45 processes per unit time

FCFS on interactive processes:

- When a process waits or blocks, it is detached from the queue and it queues up again in FCFS queue when it gets ready
- Ordering in queue may be dissimilar in second serve

Suitability and Drawbacks

- Easy to implement

- Examples: printer queues, mail queues
- Starvation free
- Suffers from Convoy Effect
- Response time

FCFS is known as non-preemptive scheduling algorithm that course once the process is allotted to the CPU, it not restart before completion if the CPU is the only schedulable resource in the system. FCFS is naturally equitable sort of everyone else is feed especially response time. Long jobs retain everyone else waiting.

Presentation of FIFO Queues

Assume number of process happening with the occurrence time zero. Consider CPU-bound process and many I/O-bound processes. As the processes flow around the system comprise two types of processes, first is CPU bound and second is I/O bound. When CPU limits process are schedules on the processor then I/O processes are waiting in the I/O queue and when the I/O bound processes are schedules the CPU limits processes are awaiting in the ready queue for scheduling that means, the CPU-bound process will get and take the CPU first. 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 indolent.

Eventually, the CPU-bound process completes its CPU burst and moves to an I/O devices. All the I/O bound processes, which have short CPU burst, accomplish quickly and move back to the I/O queues. At this time CPU sits idle. The CPU-bound process will then move back to the ready queue and be allotted 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 convey consequence as all the other processes wait for the one big processes wait for the one big process to get off the CPU. This consequence results in lower CPU and device utilization than might be possible if the shorter processes were permitted to go

first. So the FCFS scheduling algorithm is non-preemptive that means once the CPU has been allotted to a process, that process keeps the CPU until it releases the CPU, either by closing or by requesting I/O. The FCFS is thus particularly troublesome for time-sharing systems, where each user needs to get a part of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period. [1][10][12]

2.1.2 Shortest Job First (SJF)

Shortest Job First scheduling algorithm is used for minimizing response time. Shortest Job First (SJF) scheduling is provably optimal and it manufactures least process turnaround time and process average waiting time. Turnaround time is the time it takes a process from its arrival time to the time of its execution, while waiting time is the amount of time a process waits in the ready queue. SJF, however, is not practical due to its short responsiveness in Time Sharing OS. The ob problem with this algorithm is that it is need precise knowledge of how long a job or process will run and this information is not usually available and changeable. So, SJF is a dissimilar approach for the CPU scheduling in which each process with the CPU burst. If the two processes have same CPU burst then FCFS scheduling is used to resolve the problem. So it is called the shortest next CPU burst algorithm because scheduling relies on the length of the next CPU burst of a process, rather than its total length. [2][3]

The main objective of Shortest Job First (SJF) is to minimize response time .It schedules the short jobs one of the way rapidly to minimize the number of jobs waiting while a long job runs. Longest jobs do the minimum possible damage to the wait times of their competitors.

We Assume four processes arriving at time = 0, with burst time (P1 = 14, P2 =34, P3 = 8, P4 = 62) as shown in Table 2-3.

Table 2-3 SJF Performance

Pid	Reqd. Time	Waiting Time	Turnaround time	Penaly Ratio =1/Response ratio
P1	25	20	45	1.8
P2	5	0	5	1
P3	10	10	20	2
P4	5	5	10	2
Average		8.75		1.7

Throughput = 4/45 processes per unit time

Suitability and Drawbacks

- Optimal for average waiting time
- Favors shorter jobs against long jobs
- If newly arrived process are appraised at every schedule point, starvation may occur
- May not be possible to know the exact size of a job before execution

Table 2-4 SJF Scheduling

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	14
P2	0	34
P3	0	8
P4	0	62

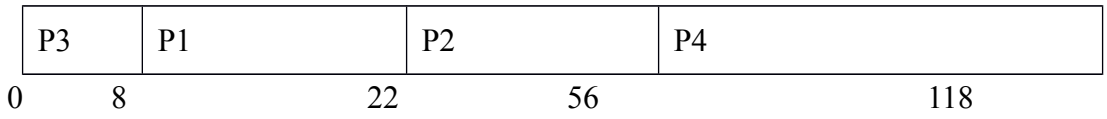


Figure 2-2 Gantt chart for SJF

Number of Context Switches = 03

Waiting time of P1 = 8

Waiting time of P2 = 22

Waiting time of P3 = 0

Waiting time of P4 = 56

Average Waiting Time = [(P1+P2+P3+P4)]/4

= (8+22+0+56)/4

= 86/4

= 21.5 ms

Turnaround Time for P1 = 22

Turnaround Time for P2 = 56

Turnaround Time for P3 = 8

Turnaround Time for P4 = 118

Average Turnaround Time = [(P1+P2+P3+P4)]/4

= (22+56+8+118)/4

= 204/4

= 51.0 ms

Performance of SJF Scheduling

With SJF, in the best-case the response time is not pretentioused by the number of tasks in the system. Shortest jobs schedule to the first. In the worst-case responsive time is unbounded, just like FCFS. The queue is not sensible because this is starvation: the longest jobs are repeatedly denied the CPU resource by the shortest job while other more new jobs continue to be fed. SJF sacrifices fairness to lower average response time. The SJF scheduling algorithm is provably optimal, in that it provides the minimum average

waiting time for a given set of processes. Moving short processes before a long one reduces the waiting time of the short process more than it creases the waiting time of the long process so consequently, the average waiting time decreases. The real problem with the SJF algorithm knows the length of the next CPU request. For long –term scheduling in a batch system, we can utilize as the length the process time limit that a user specifies when he submits the job. So, SJF scheduling is used generally in long-term scheduling. [1][11][12]

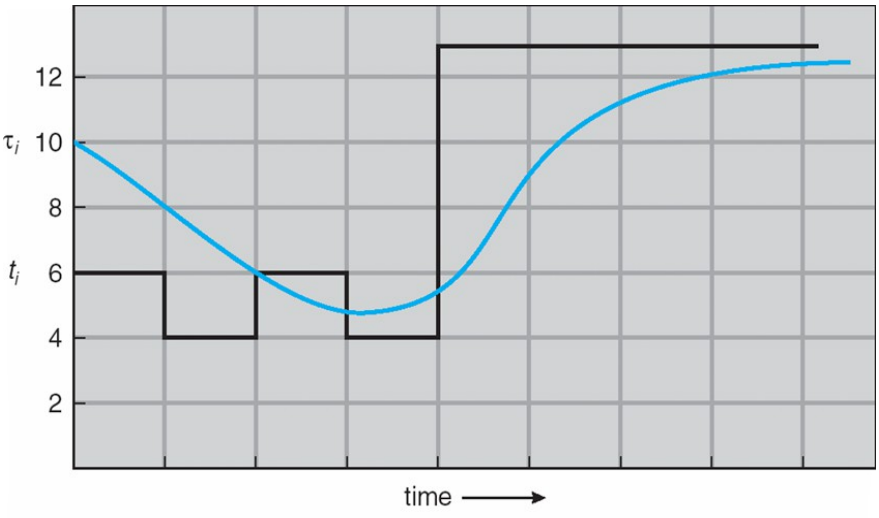
Determining Length of Next CPU Burst

- Can only estimate the length

  1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
  2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define :

- Can be done by using the length of previous CPU bursts, using exponential averaging

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 2-3 Determining Length of Next CPU Burst

Examples of Exponential Averaging

$\alpha = 0$

$\tau_{n+1} = \tau_n$

Recent history does not count

$\alpha = 1$

$\tau_{n+1} = \alpha\, t_n$

Only the actual last CPU burst counts

If we expand the formula, we get:

$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_n - 1 + \ldots$

$+(1 - \alpha)^j \alpha\, t_{n-j} + \ldots$

$+(1 - \alpha)^{n+1} \tau_0$

Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

2.1.3 Priority scheduling

In Priority Scheduling, each process is allotted with a priority, and higher priority processes are executed first, while equal priorities are executed according to the First Come First Served (FCFS) algorithm. A priority number is allotted with each process. The CPU is allotted to each process with number for assigning priority that means smaller the number higher the priority, larger the number lower the priority.

Actually SJF algorithm is the important case of priority scheduling algorithm in which we decide the priority according to the burst time that means smaller the burst time executed first. In priority scheduling allots processes to the CPU on the basis of an externally assigned priority and run the highest-priority first. The key to the performance of priority scheduling is in selecting priorities for the processes. The main difficulty of priority scheduling is starvation and the solution to this problem is aging. But the responsiveness is not good when we have multiple tasks.

Starvation

A major difficulty with Priority Scheduling Algorithm is the indefinite blocking or starvation. Starvation is a difficulty in which a higher priority job blocked the lower priority job for indefinite time that means indefinite blocking. A process that is ready to run but waiting for the CPU can be appraised blocked. A priority scheduling algorithm can leave some low priority processes waiting unlimited. In a heavily loaded computer system, a steady stream of higher priority processes can prevent a low priority process from ever getting the CPU. Generally, one of two things will occur. Either the process will finally be run, or the computer system will finally crash and lose all unfinished low-priority processes. The solution for this problem is aging. Aging we enlarged the priority of lower priority job in every 10 or 15 minutes that is the only solution for indefinite blocking.

Priorities can be described either internally or externally. Internally explained priorities use some measurable quantities to compute the priority of a process. For exemplar, time limits, memory requirements, the number of open files, and the ratio of average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the significance of the process, weight, the type and amounts of funds being paid for computer use, the department subsidizing the work.

Priority scheduling may be:

- Preemptive
- Non-preemptive

2.1.3.1 Preemptive Priority Scheduling

When a process arrives in the ready queue, its priority is contrasting with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the process if the priority of the newly arrived process is topper than the priority of the currently running process and implement the first newly running process and resumes the currently running process when higher priority process is complete the again schedules the restart process.[3]

Suitability and Drawbacks

- One can combine various parameters in one priority value
- Computing priority is a challenging task fairness must be warranted to various kinds of processes
- Tunable priorities: also from user space
- Deadlocks may happen in certain situations
- Priority Inversion problem!

Construct a deadlock case

P1 (pri=10) arrives

– P1 executes

– P2 (pri=12) arrives

– P1 is stopped and P2 executes

– Busy wait for P1

Priority Inversion

2.1.3.2 Non preemptive Priority Scheduling

A non preemptive scheduling algorithm will easily put the new process at the head of the ready queue and execute the process for the completion. It will not restart the process in mid of execution.

We Assume four processes arriving at time = 0, with random burst time (P1 = 1, P2 =10, P3 = 2, P4 = 1) with assigned priority as shown in Table2.5. Figure-2.4 shows Gantt chart for Priority Scheduling.

Table 2-5 Priority Scheduling

Process	Burst Time(ms)	Priority
P1	1	3
P2	10	3
P3	2	4
P4	1	5

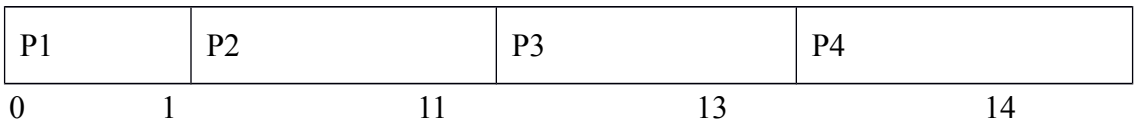


Figure 2-4 Gantt chart for Priority Scheduling

Number of Context Switches = 03

Waiting time of P1 = 0

Waiting time of P2 = 1

Waiting time of P3 = 11

Waiting time of P4 = 13

Average Waiting Time = [(P1+P2+P3+P4)]/4

= (0+1+11+13)/4

= 6.25 ms

Turnaround Time for P1 = 1

Turnaround Time for P2 = 11

Turnaround Time for P3 = 13

Turnaround Time for P4 = 14

Average Turnaround Time = [(P1+P2+P3+P4)]/4

= (1+11+13+14)/4

= 9.75 ms

2.1.4 Round Robin Scheduling: Preemptive FCFS

The Round Robin scheduling algorithm is outlined especially for Time Sharing System. Time-Sharing mentions to that sharing a computing resource among many users by means of multiprogramming and multi-tasking. By permitting a large number of users to interact concurrently with a single computer, time-sharing system give the low cost for computing capability, made it possible for individuals and organizations to use a computer for the development of new interactive applications. [3]

It is alike to First Come First Served (FCFS) scheduling, but preemption is added to switch between processes. A small unit of time quantum or time slice is explained. A time quantum range is commonly from 1 to 100 milliseconds. The ready queue is considered as circular queue when processes are scheduled in round robin manner. The CPU scheduler goes around the ready queue, allotting the CPU to each process for a time interval of up to allotted time quantum. To execute round robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue, set a timer to expire after allotted time quantum, and dispatched the process. One of the two things will then happen. The process may have a CPU burst of less than

allotted time quantum. In this case, the process itself will free the CPU voluntarily. The scheduler will then proceed to the further process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than allotted time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be happening, and the process will be put at the end of the ready queue. The CPU schedulers will the choose the next process in the ready queue and schedule it. The big disadvantage of simple round robin scheduling algorithm is that it makes higher average waiting time, turnaround time and more context switches.

Round-Robin Scheduling Algorithm is one of the easiest scheduling algorithms for scheduling processes on CPU in an operating system, which allots time quantum to each process in FCFS order and in circular order, handling all processes without priority. Round-Robin Scheduling is both simple and easy to execute, and starvation-free. In the round robin scheduling algorithm, no process is allotted to the CPU for more than allotted time quantum in a one round. If a process's CPU burst exceeds allotted time quantum that process is preempted and is put back in the ready queue. The Round Robin Scheduling Algorithm is thus preempted. With irrelevant time quantum in round robin is known as processor sharing. Round Robin Scheduling Algorithms are widely used as they give better responsiveness but poor average turnaround time and waiting time.

R = (3+5+6+e)/3 = 4.67+e

Figure 2-5 Preemptive Round Robin

Figure2-6 Evaluation of Round Robin

So the performance of Round Robin Scheduling Algorithm relies upon the size of the time quantum. The most significance issue with round robin scheme is the length of the time quantum. Setting the time quantum too short causes too many context switches, lower the CPU efficiency and the Round Robin approach is known as processor sharing creates the appearance that each of n processes has its own processor running at 1/n the speed of real processor.

On the other hand, setting the time quantum is too long may root poor response time. Now we appraise the effect of context switching on the performance of Round Robin scheduling. Let us suppose that we have only one process of 10 time units. If the time quantum is 12 time units, the process completes in less than 1 time quantum, with no overhead. If the time quantum is 6 time units, however, the process needs 2 time quantum, resulting in a context switch. If the time quantum is 1 time units, then nine context switches occur.

Process Time =10

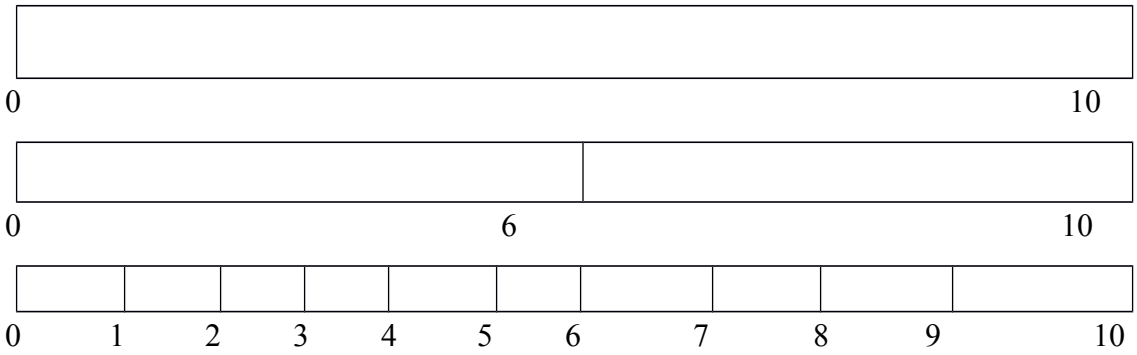


Figure 2-7 Showing Context Switches

Round-Robin Scheduling may not be relevant if the sizes of the jobs or tasks are strongly varying. This problem may be resolved by time-sharing systems i.e. by providing each job a time slot or quantum (its allowance of CPU time), and interrupt the job if it is not completed by then. Round Robin Scheduling Algorithm have some assumptions like:

- RR scheduling requires extensive overhead, especially with a small time unit.

- The average response time, waiting time is rely on number of processes, and not average process length.
- Starvation can never happen, since no priority is given. Order of time unit allotment is based upon process arrival time, similar to FCFS For e.g. the time quantum could be 100 milliseconds. If job1 takes a total time of 250ms to complete, the Round-Robin scheduler will eliminate the job after 100ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100ms each), job1 will get another allotment of CPU time and the cycle will repeat. This process continues until the job finishes and needs no more time on the CPU.

Job1 = Total time to complete 250ms (quantum 100ms).

1. First allocation = 100ms.
2. Second allocation = 100ms.
3. Third allocation = 100ms but job1 self-terminates after 50ms.
4. Total CPU time of job1 = 250ms.

In operating system, we need also to consider the effect of context switching on the performance of Round Robin Scheduling Algorithm. [1][2]

Turnaround Time Varies With the Time Quantum

Figure 2-8 Graphs between Turn around Time and Time Q

Average turnaround time

Table 2-6 Process Name and Burst Time for the Gra

Time Quantum

Process	Time
P1	6
P2	3
P3	1
P4	7

Table 2-7 Round Robin Scheduling

Pid	Reqd. Time	Waiting time	Turnaround Time	Penalty Ratio = 1/Response ratio
P1	25	20	45	1.8
P2	5	5	10	2
P3	10	20	30	3
P4	5	15	20	4
Average		15		2.7

Throughput = 4/45 processes per unit time

Suitability and Drawbacks

- Somewhere between FCFS and SJF
- Guarantees response time
- But it requires context switching

- Attempt must be made to minimize context switch time
- Process needing immediate responses have to wait for T*n-1 time units in worst case (calculate for 100 processes, 10 ms)

2.1.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been generated for situations in which processes are easily classified into different groups. For example, a common dissection is made between foreground (or interactive) processes and background (or batch) processes. These two types of processes have dissimilar response-time requirements, and so might have dissimilar scheduling needs. In addition, foreground processes may have priority (or externally defined) over background processes.

A multilevel queue-scheduling algorithm divides the ready queue into several separate queues. The processes are permanently allots to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its particular scheduling algorithm. For exemplar, 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 an FCFS algorithm. In addition, there must be scheduling among the queues, which is generally implemented as fixed-priority preemptive scheduling. For exemplar, the foreground queue may have absolute priority over the background queue.

Figure 2-9 Multilevel queue scheduling.

An example of a multilevel queue-scheduling algorithm with five queues:

1. Interactive processes
2. System processes
3. Student processes
4. Batch processes
5. Interactive editing processes

Every queue has absolute priority over lower-priority queues. No process in the batch queue, for exemplar, could run unless the queues for system processes, 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. Solaris 2 uses a form of this algorithm.

Another possibility is to time portion between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the several processes in its queue. For instance, in the foreground-background queue exemplar, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue collects 20 percent of the CPU to give to its processes in a FCFS manner.[1][40][41]

2.1.6 Multilevel Feedback Queue Scheduling

Normally, in a multilevel queue-scheduling algorithm, processes are permanently allocated to a queue on entry to the system. Processes do not move between queues. If there are individual queues for foreground and background processes, for example, processes do not proceed from one queue to the other, since processes do not change their foreground or background nature. This setup has the benefit of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to partition processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be progressed to a lower-priority queue. This scheme quits I/O-bound and interactive processes in the higher-priority queues. Similarly, a process that waits too long in a lower priority queue may be progressed to a higher-priority queue. This form of aging blocks starvation.

For exemplar, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2. The scheduler first implements all processes in queue 0. Only when queue 0 is empty will it implement processes in queue 1. Similarly, processes in queue 2 will be implemented only if queues 0 and 1 are empty. A process that appears for queue 1 will preempt a process in queue 2. A process that arrives for queue 0 will, in turn, preempt a process in queue 1.

Figure2-10 Multilevel feedback queues

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 end 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 finish, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

This scheduling algorithm provides highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, complete 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 fall 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 explained 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 also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the most general scheme, it is also the most complex

In computer science, a multilevel feedback queue is a scheduling algorithm. It is intended to meet the following design requirements for multimode systems:

1. Give preference to short jobs
2. Give preference to I/O bound processes
3. Separate processes into categories based on their need for the processor

Multiple FIFO queues are used and the operation is as follows:

1. A new process is positioned at the end of the top-level FIFO queue.
2. At some stage the process reaches the head of the queue and is assigned the CPU.
3. If the process is completed it leaves the system.
4. If the process voluntarily relinquishes control it leaves the queuing network, and when the process becomes ready again it enters the system on the same queue level.
5. If the process uses all the quantum time, it is pre-empted and positioned at the end of the next lower level queue.

6. This will continue until the process completes or it reaches the base level queue.
7. At the base level queue the processes circulate in round robin fashion until they complete and leave the system.
8. Optionally, if a process blocks for I/O, it is 'promoted' one level, and placed at the end of the next-higher queue. This allows I/O bound processes to be favored by the scheduler and allows processes to 'escape' the base level queue.

In the multilevel feedback queue, a process is given just one chance to complete at a given queue level before it is forced down to a lower level queue.

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. It is very useful for shared memory problems.

2.1.7 Real time scheduling

Why Real-Time Systems?

Due to the increasing number of the applications having timing constraints, which are too complex or fast changing to be amenable to direct human control, a need for effective real-time systems has come up.

Introduction

- “Time” is the most precious resource to be managed.
- Tasks arrive endlessly in the computer system and initiate requests for their execution.
- Every request carries a timing constraint for its completion called the “deadline”.
- The real-time task must be assigned and scheduled to be completed before its deadline.

- The correctness of computation not only depends on the logical correctness but also on the time at which results are produced.

Real-Time System

- "A real-time system may be defined as one whose principal measure of performance is whether or not it meets pre-specified task timing constraints or deadlines".
- It can also be defined as:
- "The term real-time can be used to describe any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable delay".

Components of real time systems

- Controlling System
- Controlled System

Figure 2-11 Real Time System

- The Controlled system can be viewed as the environment with which the computer interacts.
- The Controlling system interacts with its environment, acquires information through input devices that are generally sensors, performs certain computation on them, and may issue control commands to activate actuators through some controls.

Real-Time Tasks

- A real-time application usually consists of a set of cooperative tasks activated at regular intervals and/or on a particular event.
- A task typically senses the state of the system, performs certain computations, and if necessary sends commands to change the state of the system.

Real-Time Task Classification

Real-time tasks can be classified along many dimensions but the most common classification is:

- Activity Criterion: It defines how a task occurs i.e. periodically, aperiodically or sporadically
- Criticality Criterion: It defines the strictness of deadline i.e. Hard, Firm, Soft

Periodic tasks:

- Periodic tasks are time-driven and are invoked or activated at regular intervals of time called period and have deadlines by which they must complete their execution.
- The deadline of a periodic task in general may be less than, equal to or greater than the period of the task.

- Periodic tasks generally ensure the supervision of the controlled system and typically have stringent timing constraints dictated by the physical characteristics of the environment.
- Some of the periodic tasks may exist from the point of system initialization, while other may come into existence dynamically.
- Deadline of the periodic tasks must be met regardless of the other conditions in the system, failing which the system may lead to serious accidents including loss of human lives.

Aperiodic Tasks

- Aperiodic tasks are event driven and invoked only when certain events occur.
- The aperiodic tasks have irregular arrival times and are characterized by their ready time, execution time, and deadline. These values are known only when the task arrives.
- Most of the dynamic processing requirement in real-time applications is aperiodic. For example, aperiodic tasks are activated to treat errors, faults, alarms and all situations indicating the occurrence of abnormal events in the controlled system or dynamic events such as object falling in front of a moving robot or a human operator pushing a button on a console.
- These aperiodic tasks can occur at any moment in the controlled system and may have deadlines by which they must finish or start, or may have a constraint on both start and finish times.
- If the event is time-critical, the corresponding aperiodic task will have a deadline by which it must complete its execution.
- If the event is not time-critical, the corresponding aperiodic task may have a soft deadline or even no deadline, but in both the cases it must be serviced as soon as possible without jeopardizing deadlines of the other tasks.

Sporadic tasks

- Sporadic tasks are special aperiodic tasks with known minimum interarrival times and having hard deadlines.
- These are also activated in response to some external events that occur at arbitrary point of time, but with defined maximum frequency.
- Each sporadic task will be invoked repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences. For example, a sporadic task with a minimum interarrival time of 100 ms will not invoke its instances more often than each 100 ms. In fact, time between two consecutive invocations may be much larger than 100 ms.

Classification of RT Systems

- Hard Real-Time Systems
- Soft Real-Time Systems
- Firm Real-Time Systems

Hard Real-Time Systems

- Have stringent timing requirements
- Timing constraint of safety critical tasks must be satisfied under all circumstances
- The failure of computer to meet certain task execution deadline can result in serious consequences.

Soft Real Time Systems

- Have deadlines but not so stringent.
- Failure to meet soft deadlines will not result in hazardous situation.

- The utility of results produced by a task with a soft deadline decreases over time after the deadline expires.

Firm Real Time Systems

- Consequences of not meeting the deadline are not very serious.
- The utility of results produced by a task ceases to be useful as soon as the deadline expires.

Real-Time System Requirements

- Speed
- Predictability
- Reliability
- Adaptability

2.1.8 Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
- Known as process-contention scope (PCS) since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

Pthread Scheduling

API allows specifying either PCS or SCS during thread creation

- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling

- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{          int i;
           pthread t tid[NUM THREADS];
           pthread attr t attr;
           /* get the default attributes */
           pthread attr init(&attr);
           /* set the scheduling algorithm to PROCESS or SYSTEM */
           pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
           /* set the scheduling policy - FIFO, RT, or OTHER */
           pthread attr setschedpolicy(&attr, SCHED OTHER);
           /* create the threads */
           for (i = 0; i < NUM THREADS; i++)
                      pthread create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
           for (i = 0; i < NUM THREADS; i++)
                      pthread join(tid[i], NULL);
```

```
}

 /* Each thread will begin control in this function */

void *runner(void *param)

{

        printf("I am a thread\n");

        pthread exit(0);

}
```

2.1.9 Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- Processor affinity – process has affinity for processor on which it is currently running
    - soft affinity
    - hard affinity

Figure 2-12 NUMA and CPU Scheduling

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Figure2-13 Multithreaded Multicore System

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Table 2-8 Solaris Dispatch Table

Priority	Time Quantum	Time Quantum Expired	Return from Sleep
0	200	0	50

5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 2-14 Solaris Scheduling

Table 2-9 Windows XP Priorities

	Real Time	**High**	**Above Normal**	**Normal**	**Below Normal**	**Idle Priority**
Time Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above Normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Linux Scheduling

- Constant order O(1) scheduling time
- Two priority ranges: time-sharing and real-time
- Real-time range from 0 to 99 and nice value from 100 to 140

Figure 2-15 Priorities and Time-slice length

Figure 2-16 List of Tasks Indexed According to Priorities

Algorithm Evaluation

Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload Queuing models.

Figure 2-17 Evaluation of CPU schedulers by simulation

2.2 Overview of frequently used algorithms

Table 2-10 operation of Unix-like scheduling policy when processes from I/O

Scheduling algorithm	CPU Overhead	Through put	Turnaro und time	Response time

First In First Out	Low	Low	High	Low
Shortest Job First	Medium	High	Medium	Medium
Priority based scheduling	Medium	Low	High	High
Round-robin scheduling	High	Medium	Medium	High
Multilevel Queue scheduling	High	High	Medium	Medium

2.3 Operating system process scheduler implementations

The algorithm used may be as simple as round-robin in which each process is given equal time (for instance 1 ms, usually between 1 ms and 100 ms) in a cycling list. So, process A executes for 1 ms, then process B, then process C, then back to process A.

More advanced algorithms take into account process priority, or the importance of the process. This allows some processes to use more time than other processes. The kernel always uses whatever resources it needs to ensure proper functioning of the system, and so can be said to have infinite priority. In SMP (symmetric multiprocessing) systems, processor affinity is considered to increase overall system performance, even if it may cause a process itself to run more slowly. This generally improves performance by reducing cache thrashing.

2.3.1 Scheduling in UNIX

Process states and state transitions used in Unix have been described in Section A Unix process is in one of three modes at any moment:

1. Kernel non-interruptible mode
2. Kernel interruptible mode
3. User mode.

Kernel non-interruptible mode processes enjoy the highest priorities, while user mode processes have the lowest priorities. The logic behind this priority structure is as follows: A process in the kernel mode has many OS resources allotted to it. If it is allowed to run at a high priority, it will release these resources sooner. Processes in the kernel mode that do not hold many OS

resources arc put in the kernel interruptible mode.

Event addresses: A process that is blocked on an event is said to sleep on it. Unix uses an interesting arrangement to activate processes sleeping on an event. Instead of ECBs, it uses the notion of an event address. A set of addresses is reserved in the kernel for this purpose. Every event is mapped into one of these addresses. When a process wishes to sleep on an event, address of the event is computed using a special procedure. The state of the process is changed to blocked and the address of the event is put in its process structure. This address serves as the description of the event awaited by the process. When an event occurs, the kernel computes its event address and activates all processes sleeping on it.

This arrangement has one drawback-it incurs unnecessary overhead in some situations. Several processes may sleep on the same event. The kernel activates all of them when the event occurs. The processes must themselves decide whether all of them should resume their execution or only some of them should. For example, when several processes sleeping due to data access synchronization are activated, only one process should gain access to the data and other activated processes should go back to sleep. The method of mapping events into addresses adds to this problem A hashing scheme is used for mapping, hence two or more events may map into the same event address. Now occurrence of any one of these events would activate all processes sleeping on all these events. Only some processes sleeping on the correct event should resume their execution. All other processes should go back to sleep.

Process priorities: Unix is a pure time sharing operating system, so it uses the round-robin scheduling policy. However in a departure from pure round-robin. it dynamically varies the priorities of processes and performs round-robin only for processes with identical priority. Thus, in essence, it uses multilevel adaptive scheduling. The reasons for dynamic variation of process priorities are

explained in the following. A user process executes in the user mode when it executes its own code.

An interesting feature in Unix is that a user can control the priority of a process. This is achieved through the system call nice (priority value); which sets the nice value of a user process. The effective priority of the process is now computed as follows:

Process priority = base priority for user processes + f(CPU time used) + nice value;

A user is required to use a zero or positive value in the nice call. Thus, a user cannot increase the effective priority of a process but can lower it. (This is typically done when a process is known to have entered a CPU-bound phase.)

Table 2.8 summarizes operation a Unix-like scheduling policy for the processes in Table 2.9. It is assumed that process P3 is an I/O bound process which initiates an I/O operation lasting 0.5 seconds after executing on the CPU for 0.1 seconds, and none of the other processes perform I/O. The T field indicates the CPU time consumed by a process and the P field contains its priority. The scheduler updates the T field of a process 60 times a second and re-computes process priorities once every second. The time slice is 1 second, and the base priority of user processes is 60. The first line of Table 2.8shows that at t=0, only P1 exists in the system. Its T field contains 0; hence its priority is 60. Two lines are shown for t=1.0. The first line shows the T fields of processes at t=1, while the second line shows the P and T fields after the priority computation actions at t=1. At the end of the time slice, contents of the T field of P1 are 60. The decaying action of dividing the CPU time by 2 reduces it to 30, hence the priority of P1 becomes 90. At t = 2, the effective priority of P1 is smaller than that of P2 because their T fields contain 45 and 0, respectively, hence P2 it scheduled. Similarly P3, is scheduled at t = 2 seconds.

Process P3 uses the CPU for only 0.1 seconds before starting an I/O operation,

so it has a higher priority than P2 when scheduling is performed at t = 4 seconds and it gets scheduled ahead of process P2. It again gets scheduled at t = 6 seconds. These scheduling actions correct the bias against I/O-bound processes exhibited by the RR scheduler.

A process executing a system call may block for a resource or an event. Such a process may hold some kernel resources; when it becomes active again it should he scheduled as soon as possible so that it can release kernel resources and return to the user mode. To facilitate this, all processes in the kernel mode are assigned higher priorities than all processes in the user mode. When a process gets blocked in the kernel mode, the cause of blocking is used to determine what priority it should have when activated again.

Unix processes arc allotted numerical priorities, where a larger numerical value implies a lower effective priority. In Unix 4.3 BSD, the priorities are in the range 0 to 127. Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49. When a process gets blocked in a system call, its priority is changed to a value in the range 0-49 depending on the cause of blocking. When it becomes active again, it executes the remainder of the system call with this priority. When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50-127.

The second reason for varying process priorities is that the round-robin policy does not provide fair CPU attention to all processes. An I/O-bound process does not fully utilize a time slice hence its CPU usage always lags behind the CPU usage of CPU bound processes. The Unix scheduler provides a higher priority to processes that have not received much CPU attention in the recent past. It re-computes process priorities every second according to the following formula:

Process priority = base priority for user processes + f(CPU time used);

where all user processes have the same base priority.

The scheduler maintains the CPU time used by each process in its process table entry. The real time clock raises an interrupt 60 times a second, i.e., once every 16.67 msec. The clock handler increments the count in the CPU usage field of the running process. The CPU usage field is 0 to start with, hence a process starts with the base priority for user processes. As it receives CPU attention, its effective priority reduces. If two processes have equal priorities, they are scheduled in a round-robin manner.

If the total CPU usage of a process is used in computing process priority, the scheduling policy would resemble the LCN policy, which does not perform well in practice .Therefore. Unix uses only the recent CPU usage rather than the total CPU usage since the scan of a process. To implement this policy, the scheduler applies a 'decay' to the influence of the CPU time used by a process. Every second, before re-computing process priorities, values in the CPU usage fields of all processes are divided by 2 and stored back. The new value in the CPU usage field of a process is used as f(CPU time used). This way influence of the CPU time used by a process reduces as the process waits for CPU.

2.3.1.1 Fair Share Scheduling in UNIX

Unix schedulers achieve fair awe scheduling by adding one more term as follows:

Process priority = base priority for user processes + f(CPU time used by process) + f (CPU time used by process group) + nice value;

where f is the same function used in Eq. (4.5). Now the OS keeps track of the CPU time utilized by each process and by each group of processes. The priority of a process depends on the recent CPU time used by it and by other processes .

Table 2.9 depicts scheduling of the processes by a Unix-like fair share scheduler. Fields P, T and G contain process priority. CPU time consumed by a

process and CPU time utilized by a group of processes, respectively. Two process groups exist. The first group holds processes P1, P2, P3 and P5, while the second group holds process P4 all by itself. An awaited, process P4 receives a favored treatment when compared to other processes. In fact, in the period t = 5 to t = 15, it receives each alternate time slice. Processes P2, P3 and P5 suffer because they stands from the same process group. These facts are indicates in the turn-around times and weighted turn-around of the processes, which are as shown in Table 2.10.

2.3.2 Scheduling in Linux

In Linux 2.4, an O(n) scheduler with a multilevel feedback queue with priority levels varying from 0-140 was used. 0-99 is booked for real-time tasks and 100-140 are considered nice task levels. For real-time tasks, the time quantum for switching processes was around 200 ms, and for nice tasks around 10 ms. The scheduler ran across the run queue of all ready processes, letting the highest priority processes go first and run through their time slices, after which they will be set in an expired queue. When the active queue is vacant the expired queue will become the active queue and vice versa.

However, some Enterprise Linux distributions like SUSE Linux Enterprise Server replaced this scheduler with a back port of the O(1) scheduler (which was maintained by Alan Cox in his Linux 2.4-ac Kernel series) to the Linux 2.4 kernel used by the distribution.

2.3.2.1 Linux 2.6.0 to Linux 2.6.22

From versions 2.6 to 2.6.22, the kernel used an O(1) scheduler Introduced by Ingo Molnar and many other kernel developers during the Linux 2.5 development. For many kernel in time frame, Con Kolivas developed patch sets which enhanced interactivity with this scheduler or even replaced it with his own schedulers.

2.3.2.2 Since Linux 2.6.23

Con Kolivas's work, most importantly his implementation of "fair scheduling" named "Rotating Staircase Deadline", inspired Ingo Molnár to develop the Completely Fair Scheduler as a replacement for the earlier O(1) scheduler, crediting Kolivas in his announcement.

The Completely Fair Scheduler (CFS) uses a well-studied, classic scheduling algorithm known as fair queuing originally invented for packet networks. Fair queuing had been previously used to CPU scheduling under the name stride scheduling.

The fair queuing CFS scheduler has a scheduling complexity of O(log N), where N is the number of tasks in the run queue. Choosing a task can be done in constant time, but reinserting a task after it has run needs O(log N) operations, because the run queue is implemented as a red-black tree.

CFS is the first implementation of a fair queuing process scheduler broadly used in a general-purpose operating system.

The Brain Fuck Scheduler (BFS) is an substitute to the CFS.

Linux sustain both real time and non-real time applications. Consequently, it has two classes of processes. The real time processes have static priorities between 0 and 100, where 0 is the highest priority. Real time processes can be scheduled in two methods: FIFO or round-robin within each priority level. The kernel connects a flag with each process to indicate how it should be scheduled.

Non-real time processes have lower priorities than all real time processes: their priorities are dynamic and have numerical values between 20 and 19, where 20 is the highest priority. Essentially, the kernel has (100 4- 40) priority levels.

Table 2-11 Performance of fair share scheduling

Process	P1	P2	P3	P4	P5
Completion	5	13	11	14	16