

**CASE STUDY**

**CPU and Process Scheduling in Windows and Linux OS**



**Name** – S VINAY

**Subject** – Operating Systems

**What is CPU Scheduling?**

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

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.

## **Types of CPU Scheduling**

CPU scheduling decisions may take place under the following four circumstances:  
1) When a process switches from the **running** state to the **waiting** state (for I/O request or invocation of wait for the termination of one of the child processes).  
2) When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs).  
3) When a process switches from the **waiting** state to the **ready** state(for example, completion of I/O).  
4) When a process **terminates**.

In circumstances 1 and 4, 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 in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

### **Non-Preemptive Scheduling**

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

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

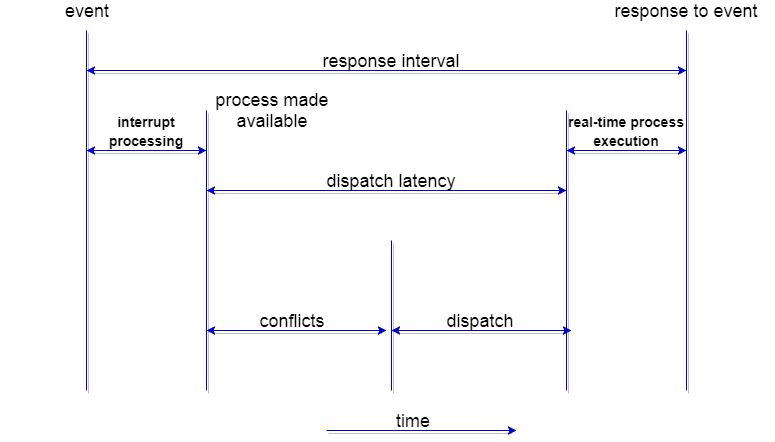
It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

### **Preemptive Scheduling**

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

## **CPU Scheduling: Dispatcher**

Another component involved in the CPU scheduling function is the **Dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the **short-term scheduler**. This function involves:  
1) Switching context  
2) Switching to user mode  
3) Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure: 

## **CPU Scheduling: Scheduling Criteria**

There are many different criterias to check when considering the **"best"** scheduling algorithm, they are:

#### CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time (Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

#### Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

#### Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process (Wall clock time).

#### Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

#### Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

#### Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

**What is Process Scheduling?**

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

**Process Scheduling Queues**

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

* **Job queue** − This queue keeps all the processes in the system.
* **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
* **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

**Schedulers**

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler

**Comparison among Scheduler**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Long-Term Scheduler** | **Short-Term Scheduler** | **Medium-Term Scheduler** |
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long-term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

**Context Switch**

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

**SCHEDULING IN WINDOWS OS**

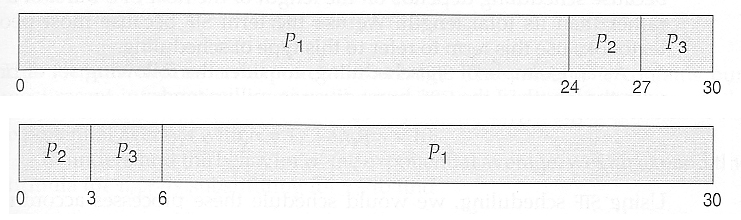
1. FIRST COME FIRST SERVE(FCFS)
2. SHORTEST JOB FIRST(SJF)
3. ROUND ROBIN(RR)
4. PRIORITY
5. MULTI LEVEL QUEUE SCHEDULING
6. MULTI LEVEL FEEDBACK QUEUE SCHEDULING

#### **First-Come First-Serve Scheduling, FCFS**

* FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
* Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

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

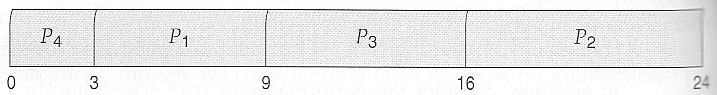
* In the Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.



#### **Shortest-Job-First Scheduling, SJF**

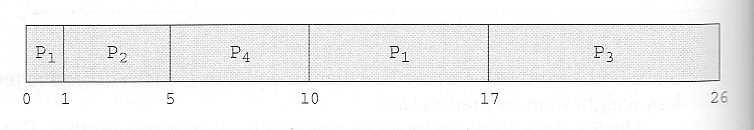
* The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
* (Technically this algorithm picks a process based on the next shortest**CPU burst**, not the overall process time.)
* For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |



* In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)
* SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?  
  1)For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However, that does not work for short-term CPU scheduling on an interactive system.  
  2) Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.
* SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling.***
* For example, the following Gantt chart is based upon the following data:

|  |  |  |
| --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** |
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| p4 | 3 | 5 |

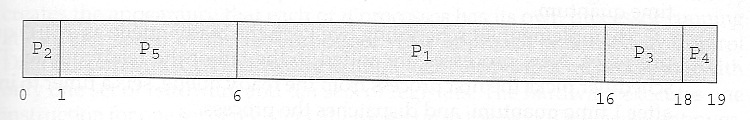


* The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

#### **Priority Scheduling**

* Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
* Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
* For example, the following Gantt chart is based upon these processes burst times and priorities, and yields an average waiting time of 8.2 ms:

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

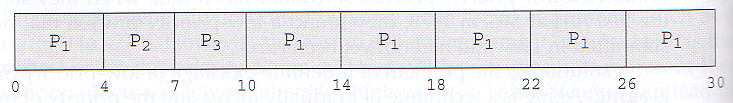


* Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
* Priority scheduling can be either preemptive or non-preemptive.
* Priority scheduling can suffer from a major problem known as ***indefinite blocking***, or ***starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
  + If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumours of jobs that have been stuck for years.)
  + One common solution to this problem is ***aging***, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

#### **Round Robin Scheduling**

* Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum***.
* When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
  + If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
  + If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
* The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
* RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms.

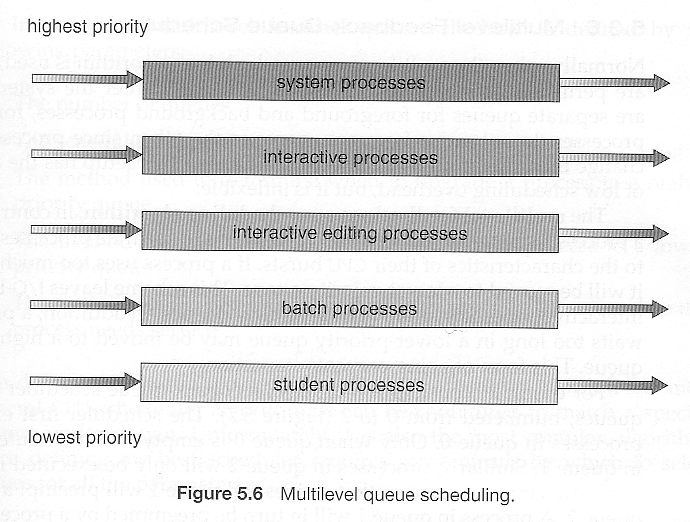
|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |



* The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.

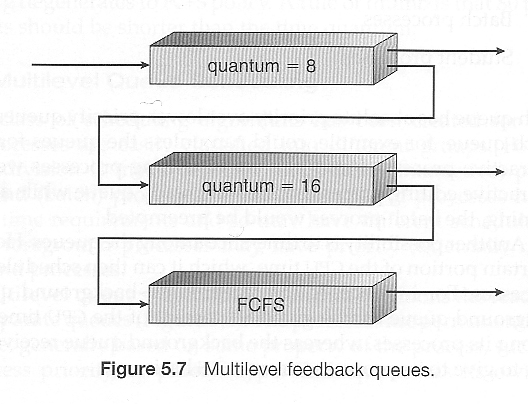
#### **Multilevel Queue Scheduling**

* When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
* Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
* Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.



#### **Multilevel Feedback-Queue Scheduling**

* Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  + If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  + Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.
* Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
  + The number of queues.
  + The scheduling algorithm for each queue.
  + The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  + The method used to determine which queue a process enters initially.



**SCHEDULING IN LINUX OS**

**Linux 2.4**

In Linux 2.4, an O(n) scheduler with a multilevel feedback queue and priority levels ranging from 0 to 140 is used. 0–99 are reserved for real-time tasks and 100–140 are considered nice task levels. For real-time tasks, the time quantum for switching processes was approximately 200s, and for nice tasks approximately 10 ms. The scheduler ran through 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 placed in an expired queue. When the active queue is empty the expired queue will become the active queue and vice versa.

**Linux 2.6.0 to Linux 2.6.22**

In those versions, the kernel used an O (1) scheduler, a kernel scheduling design that can schedule processes within a constant amount of time, regardless of how many processes are running on the operating system. O (1) scheduler providing "constant time" scheduling services has helped to minimize overhead and jitter of OS services.

 **Since Linux 2.6.23**

Those Linux versions implemented a Completely Fair Scheduler as a replacement for the earlier O (1) scheduler.

The Completely Fair Scheduler (CFS) uses a scheduling algorithm called fair queuing. Fair queuing had been previously applied 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 requires O (log N) operations, because the run queue is implemented as a red-black tree.

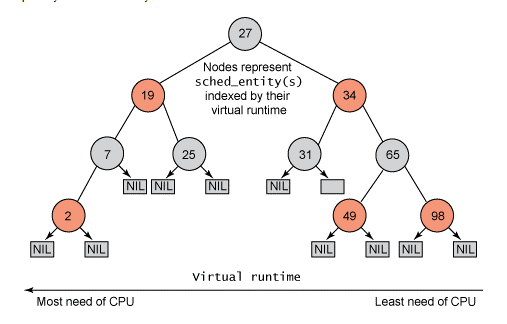
CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.

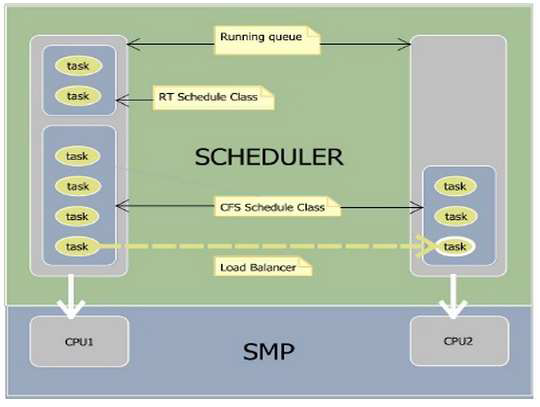
**Completely fair scheduler**

The main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. This means processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute.

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the virtual runtime. The smaller a task's virtual runtime—meaning the smaller amount of time a task has been permitted access to the processor—the higher its need for the processor. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it.

But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time-ordered red-black tree (see Figure below). A red black tree is a tree with a couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in O (log n) time (where n is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently

**Concrete view of Linux Kernel Scheduler (LINUX 2.6)**



**Linux scheduler contains:**

 A Running Queue: A running queue (rq) is created for each processor (CPU). Each rq contains a list of runnable processes on a given processor.

Schedule Class: It is an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are called from the scheduler core without the core code assuming too much about them. There are two schedule classes implemented:

 Completely Fair Schedule class: schedules tasks following Completely Fair Scheduler (CFS) algorithm. Tasks which have policy set to SCHED\_ NORMA L

(SCHED\_OTHER), SCHED\_BATCH, SCHED\_IDLE are scheduled by this schedule class.

 RT schedule class: schedules tasks following real-time mechanism defined in POSIX standard. Tasks which have policy set to SCHED\_FIFO, SCHED\_RR are scheduled using this schedule class.

 Load balancer: In SMP environment, each CPU has its own rq. These queues might be unbalanced from time to time. A running queue with empty task pushes its associated CPU to idle, which does not take full advantage of symmetric multiprocessor systems. Load balancer is to address this issue. It is called every time the system requires scheduling tasks. If running queues are unbalanced, load balancer will try to pull idle tasks from busiest processors to idle processor.

**Real time scheduling scheduler**

A real-time system is one that provides guaranteed system response times for events and transactions—that is, every operation is expected to be completed within a certain rigid time period. A system is classified as hard real-time if missed deadlines because system failure and soft real-time if the system can tolerate some missed time constraints.

Linux has soft real-time scheduling, Processes with priorities [0, 99] are real-time and all real-time processes are higher priority than any conventional processes.

There are two real-time scheduling systems, FCFS and round-robin: First-come, first-served; process is only preempted for a higher-priority process; no time quanta.

Round-robin; real-time processes at a given level take turns running for their time quantum.

**Scheduling techniques (algorithms)**

The Linux scheduler tries to be very efficient, and uses different algorithm to manage processes. The main purpose of scheduling algorithms is to minimize resource starvation and ensure fairness amongst the parties using them. Among them we can distinguish:

1. **First Come First Served (FCFS):** A simple non preemptive real time scheduling algorithm based on the arrival order to the ready queue.

a. Overhead is minimal (context switch occurs only after process completion)

b. Throughput can be low, waiting and running time can be low: long process can hold the CPU.

c. No prioritization, thus this system has trouble meeting process deadlines.

d. No prioritization means that as long as processes are terminating, there is no starvation, but in an environment where some processes might not complete, then starvation can occur.

e. Based on queuing.

2. **Shortest Job First (SJF):** In this approach, the running process is selected regarding its burst time. The smallest burst time process presents in the ready Queue takes on the CPU. This algorithm can be preemptive or non-preemptive (in preemptive the remaining time is calculated when a new process arrives, hence shortest Remaining time first (SRTF)).

This algorithm is optimal since it gives the minimal average waiting time, but present some drawback point, since it requires to know the length of the next CPU burst of a process. Moreover, it can cause starvation of long processes if shorter ones keep entering the ready queue.

3. **Priority Scheduler:** a non-preemptive algorithm where each process is assigned a priority. Process with the highest priority gets the CPU and if two processes are having the same priority then execution is FCFS based. Its biggest drawback is starvation of low priority processes, in order to go over this problem, dynamic priorities might be used (priority of an executed process is decreased after its CPU burst). Another approach is to have the CPU keep track of low priority processes and increase their priority (once executed its priority go back to lower one)

 Dynamic priority is calculated from static priority and average sleep time.

When process wakes up, record how long it was sleeping, up to some maximum value then when it is running, decrease that value each timer tick.

Roughly speaking, the bonus is a number in [0, 10] that measures what percentage of the time the process was sleeping recently; 5 is neutral, 10 helps priority by 5, 0 hurts priority by 5 DP = max (100, min (SP − bonus + 5, 139)).

4. **Round Robin:** A preemptive scheduling technique that is based of fair sharing of CPU time (time sharing). In this technique fixed time (quantum) is allocated to all the processes on the ready queue, so that the CPU cycle through them and no process remains in the ready queue for long time (starvation free).

5. **Multilevel Queue:** This is used when processes can be divided into foreground (interactive) processes and batch (background) processes. In this scheduling algorithm, there is 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like Round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the processes in those high priority queues should have completed their execution. In this scheduling algorithm, once assigned to a queue,

the process will not move to any other queues. Foreground processes are given high priority and background processes have low priority.

**Conclusion**

Through this work most of the important aspects of Linux scheduler has been covered. Kernel scheduler is one of the most frequently executed components in Linux system.

Hence, it has gained a lot of attentions from kernel developers who have thrived to put the most optimized algorithms and codes into the scheduler.

Different algorithms used in kernel scheduler were discussed in the project. CFS scheduler achieves a good performance and responsiveness while being relatively simple compared with the previous algorithm like O (1).

CFS exceeds performance expectation in some workloads. But it still shows some weakness in other workloads.

Kernel scheduling is in constant improvement, developers always seeking for better performance and simpler algorithms.

Well for now this present work allowed us to understand Linux scheduling logics and improvement through time.