A Study of Available Process Scheduling Algorithms and Their Improved Substitutes

Sameer Mahajan[1], Adwait Patil[2], Nahush Kulkarni[3]

[1,2,3]*Student, Department of Computer Engineering, TEC, University of Mumbai, Mumbai, India*

Abstract: *Process scheduling is an essential part for any modern operating system to ensure efficient utilization of the CPU. Several scheduling algorithms have been in use for decades and are being enhanced actively to increase their efficiency. To contribute to this development, we have studied the fundamentals of process scheduling in Real-Time and Distributed systems. We have discussed the algorithms currently being used by various operating systems, and have examined their improved substitutes.*

Index Terms: First-Come-First-Serve, Highest Response Ratio Next, Multilevel Feedback, O (1) Scheduler, Process Scheduling, Real-Time, Round Robin.

1. INTRODUCTION

An operating system, being an interface between the user and hardware, has several responsibilities. It has to manage processes, storage, network, devices and a lot more. In this paper we are going to discuss about the management of processes, also known as Process Scheduling.

System resources such as bandwidth, communication, processor cycles and many more are made accessible to the processes on a system through process scheduling. Almost every modern system is capable of running several processes at the same time. This is called multitasking. There are several ways to run these processes on while performing efficient utilization of resources. Most common of which have been mentioned in section 2. [1]

1.1 Types of System

Depending upon the use cases, there can be several types of computer systems. Here, we are going to discuss about the following types:

1. Real-Time systems: In real-time systems, time is the most important factor to be considered. These systems have to respond within the given time constraints and deadlines.
   a. Soft Real-Time Systems: In these systems, deadlines can be missed occasionally.
   b. Hard Real-Time Systems: In these systems, deadlines cannot be missed. Missing of deadlines can have disastrous consequences. [2]
2. Distributed Systems: In Distributed Systems, a set of computers is connected to each other by a network. Scheduling problems of these systems can be solved using several proposed methods.
   These methods can be classified as:
   a. Mathematical Models
   b. Graph-Theory Based Methods
   c. Heuristic Methods
      i) Constructive
      ii) Iterative
      iii) Probabilistic [3]

1.2 Scheduling paradigms for Real-Time Systems:

Several approaches can satisfy the needs of a real-time system. Most common of which are:

1. Static Priority Driven Preemptive Approach: In this approach, static scalability is analysed but construction of explicit schedule do not occur.
2. Static Table-Driven Approach: Just like the above method, static scalability is analysed but, the resulting schedule is considered explicit.
3. Dynamic Best Effort Approach: In this approach, feasibility is not checked and the deadlines are met by the best tries of the system. A task might be stopped during its execution as no guarantees are provided.
4. Dynamic Planning-Based Approach: Unlike the above static approaches, a feasibility check is done at runtime. [4]

1.3 Role of Machine Learning:

How a process runs depends a lot on how it has been coded. Hence, prediction of time required to run a process can be very helpful. For example, if a process has some of its part left to be executed, it should not be pre-empted as this can increase the number of context switches. Advancements in the field of machine learning has played an important role in the improvement for these predictions. Scheduling behaviours can now be very much predicted. However, it is important to note that the results from these techniques depend upon the dataset on which the ML technique was trained. [5]

1.4 Designing Scheduling Algorithms:

Designing and selection of proper scheduler is an important task. Better utilization of resources is always good. This is also a problem most developers face. A programmer has to take some factors into consideration while developing a scheduler.

Following are key parameters for any scheduler:

1. Turnaround Time (TAT): It is the time taken for the completion of a job after its submission.
2. Waiting Time (WT): It is the time a process has to wait in ready queue before its execution.
3. Context Switch: In a preemptive multitasking system, a job is stored in memory to allocate CPU resources to another process. This stored job can then be restored at a further point in time so resume its execution.
4. Throughput: It is the number of jobs a system completes per unit time.
5. CPU Utilization: It is the amount of work the CPU is doing. [1]

A good scheduler is the one which has:

1. Less Turnaround Time,
2. Less Waiting Time,
3. Less Context Switches,
4. High Throughput,
5. High CPU Utilization. [1]

2. AVAILABLE SCHEDULING ALGORITHMS

1. Round Robin (RR): Mainly implemented in time-shared systems, this is the most basic algorithm for scheduling. A time-slice or quantum, being the time for which a CPU is allocated to a process is used. Processes are added to queue in FCFS manner. Every process is kept in a circular queue and executed when the ones prior to it have completed their quantum. This method is best suited for processes with a short burst time.

2. First Come First Serve (FCFS): In this approach, the process that enters the ready queue first gets executed first.

3. Shortest Job First (SJF): As the name suggests, jobs with shortest burst or remaining time are executed first. In SJF Non-Preemptive, ascending order of burst times is followed for execution on processes. In SJF Preemptive, burst time of processes is checked after every unit in time. This is also called, Shortest Remaining Time First.

4. Longest Job First (LJF): Just opposite to SJF, larger burst time processes are executed first.

5. Highest Response Ratio Next (HRRN): Waiting time of a process is used to boost the priority of the process. This is also known as the "aging priority" schema. Below is the formula used to calculate the priority of a process. Here, s is the expected time of service and W is the time spent waiting by the process.

$$Priority(HRRN) = \frac{(W + s)}{s}$$

6. Priority-Based: In this approach, processes are put to the ready queue based on the priority number provided to the process by the operating system.

7. Multilevel Queue: In this approach, loads are divided into different queues. Different scheduling criterions are used for different processes. Priority of a process is used to determine where process will be added in the ready queue. Since higher priority processes are placed at the top of the queue, starvation might be experienced by processes with lower priorities.

8. Multilevel Feedback: To solve the problem of starvation from Multilevel Queue Scheduler, processes are placed to the next level if their execution has not finished at a particular level.

9. Job Mix: Unlike any other approach, processes with shorter burst times are executed first while using a different queue for keeping processes with higher burst times. This method can solve problems of starvation but at the cost of increased overhead due. This method is very easy to implement.

10. Standard Deviation (SD): Using the formula mentioned below, the processes with burst time nearer to the standard deviation value are placed into the queue. This process is continued until all the processes have been placed into the queue. In the below formula, N is the number of processes, X is the burst time, and $\bar{X}$ denotes the average burst time of all processes. [1]

$$SD = \sqrt{\frac{\sum(X - \overline{X})^2}{N}}$$

3. PROCESS SCHEDULING ALGORITHMS CURRENTLY USED BY OPERATING SYSTEMS

1. Round-robin algorithm with multilevel feedback queue (used by Windows)
2. O (1) scheduler algorithm (used by Android)

3.1 Round Robin algorithm

As described in section 2, Round robin is a preemptive process scheduling algorithm. Here the quantum should

not be too small as it leads to wastage of CPU time in process switching. Process Scheduling of Round Robin is as follows:

1. To schedule process fairly round-robin scheduler employs time-sharing, that is it assigns quantum to each job, and it stops the execution of the job if it's not completed in that quantum
2. If the process under execution terminates or changes its state to waiting then the algorithm stops execution of this process and allots time to the first process in the ready queue.
3. Round robin is called preemptive as it forces a process to stop once it has used its quantum.
4. Example: If there is a job J which requires 350 ms to complete and the algorithm has a quantum=100 ms then
   a. First allocation = 100ms (algorithm preempts the job)
   b. Second allocation = 100ms
   c. Third allocation = 100ms
   d. Fourth allocation = 50ms (job terminates itself)
   e. Total time for J = 350ms

Table -1: Details of Processes for Example

Process name	Arrival time	Execution time(ms)
P0	0	120
P1	25	270
P2	130	80
P3	210	190
P4	260	100

Consider table 1 with arrival time and execution time of various processes. Quantum = 100ms.

Table -2: Round Robin Execution Example

Time	Executing	Waiting	Comments
0	P0	-	P0 arrives and gets processed
25	P0	P1	P1 arrives and waits as P0 is still executing
100	P1	P0	Quantum time for P0 expires so its terminated and P1 is processed
130	P1	P0, P2	P2 arrives and waits
200	P0	P2, P1	P1 is terminated and P0 starts executing
210	P0	P2, P1, P3	P3 waits
220	P2	P1, P3	P0 self terminates as its execution time ends (120ms)
260	P2	P1, P3, P4	P4 arrives and waits
280	P1	P3, P4	P2 terminates itself as its execution time ends (80 ms)
380	P3	P4, P1	P1 terminates, P3 is processed
480	P4	P1, P3	P3 terminates as its quantum is exhausted and P4 is processed
580	P1	P3	P4 terminates as its execution time ends(100ms)
650	P3	-	P1 terminates itself as its execution time ends(100+100+70=270ms)
740	-	-	P3 terminates itself as its execution time ends(100+90=190ms)

Table 2 shows how these process are handled by round robin scheduling with quantum 100ms.

3.2 O(1) scheduler algorithm

The O(1) scheduler was introduced in 2.6 Linux kernel. The scheduling done after implementing O(1) scheduler always completes in a constant time and hence its aptly named O(1) scheduler indicating the constant time. This approach helps in reducing the total system time and thus makes processing more efficient. The O(1) scheduler was created in order to achieve some specific objectives which are as follows

1. Implement an algorithm that always completes in constant time independent of the number of tasks it is working on.
2. The algorithm should provide a consistent performance for interactive tasks even if the system is under load.
3. To be fair to all the running processes irrespective of arrival time or execution time and thus preventing starvation.
4. Provide optimum or consistent processing for few number of processes but also maintain the performance in a multiprocessor system.

The data structures used are designed to assist the algorithm in achieving the above mentioned goals. The data structures are as follows:

1. Runqueues: This is one of the basic data structures in O(1) scheduler defined as struct runqueue. It gives the count of processes waiting for execution on a given processor, every processor has a runqueue. The runqueue also contains pre-scheduling information.

2. Priority arrays: There are 2 priority arrays present in each runqueue, one array for keeping count of active processes called active array and another for keeping count of terminated processes called expired array. This data structure is the one that provides O(1) scheduling also called constant time scheduling. Each priority array contains one queue which contains the number of runnable processes ordered according to their priority. Priority arrays also contain a bitmap which can be used to track down the process which has the highest priority in the queue.
3. Process descriptor: A process descriptor has several fields that play a key role in scheduling, such as:
   a. need_reshed: This is actually a flag which is used to decide whether the scheduling() function should be invoked or not depending on the value of this flag.
   b. rt_priority: This function returns the static priority of a process. The default value for static priority of any process in the system is zero but it can be changed.
   c. counter: The CPU-time left before a quantum expires is tracked using counter. When a new process begins it holds the time-quantum of that process.
   d. nice: This is a crucial function present in the process descriptors as it is used to change the static priority of a process. This function can set the static priority in a range (-19,+20) the more the value of static priority tends to negative higher the priority of that process.

Calculating priority and time slice in O(1):

1. Static task prioritization using nice(): All processes have a default static priority of 0 but this priority can be changed using the nice() function mentioned above in process descriptors. The nice() function can be used to change this default value to any value ranging between -20 and +19.
2. Prioritizing tasks dynamically: The O(1) scheduler is biased towards I/O bound tasks and rewards them whereas it punishes CPU bound tasks by adding or subtracting static task priority using nice(). This new priority is called the dynamic priority of a task and can be found in a task's priority variable. In O(1) scheduler, the maximum reward and the maximum penalty is equal to 5.
3. I/O bound vs CPU bound processes: The processes are awarded or penalised depending upon the value of a variable called sleep_avg. The value of this variable depends on the sleep time of a process. When a task comes out of sleep, the total time for which the task was sleeping is calculated and this time is appended on to the sleep_avg variable. When a task gets CPU time then the time for which it runs is subtracted from sleep_avg. The higher the value of sleep_avg higher is the dynamic priority.
4. The effective_prio() function: The effective_prio() function is used for calculating the dynamic priority of a process. This function takes a task's sleep_avg as a parameter to calculate its dynamic priority.

```
bonus = CURRENT_BONUS(p) – MAX_BONUS / 2;
prio = p->static_prio – bonus;
//CURRENT_BONUS is defined as:
#define CURRENT_BONUS(p)
NS_TO_JIFFIES((p)->sleep_avg)* MAX_BONUS /
MAX_SLEEP_AVG;
```

CURRENT_AVG maps a task's sleep_avg onto the range 0 to MAX_BONUS. And as MAX_BONUS is twice that of the maximum possible reward or penalty it is divided by two before subtracting from CURRENT_BONUS.
Example: 1. If a process has high sleep average and CURRENT_BONUS returns 10 then according to the formula mentioned above bonus = 10 – 10/2. (As sleep_avg is high MAX_BONUS=10). bonus = 5. And thus prio gets reduced by 5 which is the maximum reward (priority increases as prio tends to negative values) [6]

3.3 Multilevel Feedback queue algorithm

As described in section 2, in a multilevel queue scheduling algorithm, processes are permanently assigned to one of the queues on entry to the system. Processes cannot be transferred from one queue to another. This approach gives it an advantage of low scheduling combinations that is it has no excess combinations which can reduce efficiency of the algorithm but also has a disadvantage that this makes it inflexible. These drawbacks can be nullified using a multilevel feedback queue which enables the processes present in the queues transfer from higher-level queue to lower-level or vice versa depending upon the method used by the algorithm. It relies on a basic idea of separating processes based on the CPU-time they use during execution. If a process is using high amount of CPU time then that process is demoted to a lower-level queue with larger time quantum. Tantamount to this if a process has a high waiting time then there is a possibility that it can be promoted to a higher level queue which has a smaller time quantum so the process gets CPU time faster.

The multilevel feedback queue's efficiency depends upon the following factors:

1. The total number of queues(top-level to base-level) present in the algorithm.
2. The algorithm used for scheduling each individual queue.

3. The logic that is implemented while promoting a queue to higher-priority.
4. The logic that is implemented when a process is demoted to lower-priority.

Thus as we can make a multilevel feedback queue to work efficiently for any system by altering the factors it depends on which makes it one of the most general CPU scheduling algorithm. But it also needs some means to select ideal values of the factors for creating the most efficient scheduler for a particular system. This makes the multilevel feedback queue one of the most general as well as the most complex scheduling algorithm.

It uses multiple FIFO (First In First Out) queues to place the processes entering the system. The operations done by the algorithm are:

1. Whenever a new process enters into the system it is always appended at the end of the top-level queue.
2. After that when the process gradually reaches the head of the top-level queue it is allotted CPU time and gets executed for the time quantum of that queue.
3. If the process either terminates or goes to sleep during execution then it leaves the queue it was inserted into.
4. If the process terminates itself amidst the time quantum of the given queue, then it leaves the queue network and when it again becomes active and is ready to get executed it is again appended at the end of the same queue which it belonged to.
5. If the process exhausts its given time quantum then it is pre-empted and appended at the end of a lower-level queue in the network. Any lower-level queue always has a time quantum exceeding the time quantum of the queue which has higher priority. Thus following this structure a process keeps getting demoted every time it exhausts the time quantum of its present queue until it reaches the base level queue.
6. After reaching the base level queue the processes undergo round robin algorithm until the time they get terminated. Optionally processes at base level queue may also undergo FCFS (First Come First Serve) depending on the time quantum or if required for efficiency of certain system.
7. If a process is I/O bound or waiting for I/O device it is given higher priority and is promoted to the preceding higher level queue. Thus the algorithm is biased for I/O bound processes and lets them avoid staying in base level queue.
8. The algorithm always allots CPU time to the task present in the head of the top-level queue. And only after the top-level queue becomes empty it shifts to the next lower-level queue and this pattern keeps repeating until it reaches base level.

A new process is always appended at end of top-level queue assuming it will end in short time-quantum or else its pushed to lower level queue. Thus every process gets just one chance to get executed in a given queue.

4. ADVANTAGES AND DISADVANTAGES OF PROCESS SCHEDULING ALGORITHMS

The process scheduling algorithms currently used are powerful and nearly flawless but still there are some limitations faced even by these powerful algorithms and those are as follows:

4.1 Round Robin algorithm

Advantages:

1. The round robin algorithm executes or allots CPU time to each process for a particular quantum of time thus all processes get same priority.
2. In a round robin cycle all processes get execution time and hence this prevents the processes from starving for CPU time.

Disadvantages:

1. The efficiency of round robin highly depends on the size of the time quantum if the time quantum is comparatively large then it acts similar to FCFS thus including all its drawbacks.
2. If a time quantum is comparatively very short it sure does prevent starvation but the context switching increases in frequency leading to reduced efficiency of the CPU.
3. Average waiting time is often longer in Round Robin.

4.2 Multilevel feedback queue algorithm

Advantages:

1. As the quantum of time increases as we go down from top-level queue to base-level queue processes with large execution time gets executed in a single quantum in lower-level queue thus reducing context switching time of the CPU.
2. If a process is I/O bound or waiting for I/O device it is given higher priority and is promoted to the preceding higher level queue thus it is allotted CPU time faster preventing starvation.

Disadvantages:

1. Moving the processes around the queue increases CPU overhead.
2. It needs some means to select ideal values for all the factors in order to define the most efficient scheduler for that particular system. Thus making it one of the most complex scheduling algorithm.

4.3 O (1) Scheduling algorithm

Advantages:
1. The O(1) scheduler executes in constant time regardless of the number of tasks running
2. It provides a consistent interactive performance even when system is under load.

Disadvantages:
1. The O(1)n scheduler is bias towards I/O bound tasks and assigns rewards to them while they sleep for longer time thus they gain higher priority and execute first ,thus the lower priority CPU bound tasks starve for CPU time.

5. REFERENCED ALGORITHMS

5.1 Introduction

Tahhan et. al [7] proposed a priority algorithm using the knapsack algorithm. The Knapsack problem consists of number of elements, the weight and value of the elements, the problem is to determine the number of elements with a total weight less than or equal to the knapsack capacity, with a condition that the total value is as high as possible. The proposed NOPSACK algorithm was as follows [7]:
Input:
   a. No. Of Processes(n)
   b. CPU execution time(exet)
   c. Priority(P)
1. Calculate ratio (P/exect) for each process.
2. Arrange processes in descending order of the ratio.
3. Calculate initial (ß) value (sum of execution time of all the processes(α)/No. of processes(n) in the system).
4. Apply knapsack algorithm:
   Knapsack elements = No. Of Elements(n)
   Weight = CPU execution time(exet)
   Cost = Priority(P)
5. Invert the priority for the process in the current iteration, ((Pri) value * (-1)). Also cancel the processes which are completed.
6. Go-to step 3 (continue until all processes got CPU and completed execution).
7. Display and Print result of each process

The results were that the Average Turnaround Time (TAT) and Avg Waiting (WT) obtained by NOPSACK Algorithm were better than Round Robin Algorithm, Priority Based Algorithm, and Neelsack Algorithm. [8]

Nie et. al [9] compared various algorithms such as Shortest-Job (process)-First, First-Come, First-Served, Highest Response Ratio Next. They proposed median-time slice-Highest Response Ratio Next algorithm. The proposed algorithm is as follows [9]:

1. Sort all the processes according to their respective execution time (exet)
2. Calculate Median of the processes.
3. Scan all the process execution time, if the process execution time larger than the median, then divide it into several parts according to median, each integer multiples as a time slice, the mod as a time slice.
4. Each part of the PETLM and the PETSM as the new process queue, carry out HRRN.
5. Calculate the RR
6. Sort to the RR, the largest allocate the CPU; mark the process or the sub process had holder.
7. Go to step 1, till all process or the sub-process have holder.

The proposed algorithm was compared with various algorithms like FCFS, SJ (P) F, HRRN and was observed feasible and effective against them.

Kolipakula et. al [1] proposed a hybrid scheduling algorithm using Standard Deviation and Highest Response Ratio Next (HRRN). The following is the proposed algorithm [1]:
1. Read process Id, Arrival time and Burst time for all the processes
2. Calculate Standard Deviation using formula:

$$SD = \sqrt{\frac{\sum(X - \overline{X})^2}{N}}$$

3. Calculate Highest Response Ratio First using the formula:

$$Priority(R) = \frac{(W + s)}{s}$$

4. Calculate Hybrid Priority using formula:

$$Hybrid\ Priority(IIP) = (0.5 * R') + (0.5 * SD)$$

5. If two processes having same hybrid priority value?
   If true, did processes having same hybrid priority value arrives at a particular time?
   If true, the process with lower process id will be executed first.
   Else, did processes having same hybrid priority value arrives at different time?
   If true, the process with less arrival time will be executed first.
6. Execute process according to its Hybrid Priority
7. Process leaves queue if its burst time is zero. Calculate waiting time and turnaround time of the process.
8. If queue is empty, then calculate average waiting time, average turnaround time and average response time of all the processes.

It was observed that the proposed Hybrid Scheduling algorithm is very efficient over the other rescheduling algorithms in parameters of average turnaround time and average waiting time.

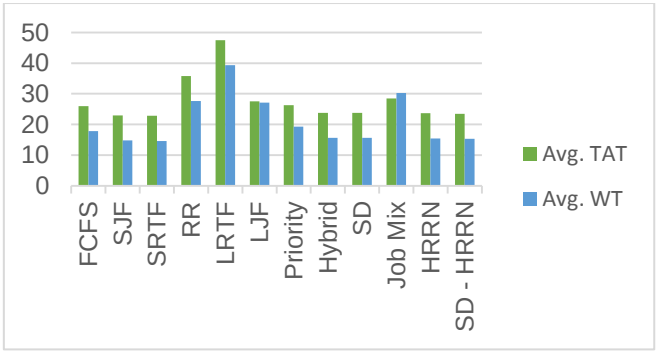

Fig -1: Performance of SD-HRRN

Nikravan et. al [3] proposed a genetic scheduling algorithm for Distributed Computing System (DCS). The algorithm is [3]:

1. Generate initialize population
2. Evaluate all individuals in that population.
3. For i=1 to 2*POPSIZE do
   Select two chromosomes from the population.
   (Parent 1 and Parent 2)
   and parent 2 from population;
   Child 1 and Child 2 ← Crossover (parent1, parent2);
   Child 1 ← Mutation (Child 1);
   Child 2 ← Mutation (Child 2);
   Add (new temporary population, Child 1, Child 2
4. End For;
   Make (new population, new temporary population, population);
   Population = new population;
5. While (not termination condition);
6. Select Best chromosome in population as solution and return it;
7. End

5.2 Comparison of Performance:

Table -3: Performances of Referenced Algorithms

Paper	Algorithms	Avg. TAT	Avg. WT
Kolipakula et. al [1]	Round Robin	30.66	22.5
	FCFS	22.67	14.5
	SJF	20.33	12.33
	HRRN	24.33	16.16
	Priority-Based	26.33	18.16
	Hybrid Algorithm	21.16	13
Tahhan et. al [7]	Round Robin	15.7	10.1
	Priority-Based	15.8	10.1
	NOPSACK	11.98	6.8
Nie et. al [9]	FCFS	2.84	4.92
	SJF	1.92	2.23
	HRRN	2.5	2.11
	MTSHRRN	2.9	2.52

Observing the results in Table 3, Hybrid algorithm[1] is surely better based upon the Turnaround Time and Waiting Time of Round Robin and Priority-Based algorithms when compared to NOPSACK[7] and also, than MTSHRRN[9] based on the results of FCFS, SJF and HRRN.

6. CONCLUSION

From the study conducted, we can conclude that process scheduling being an important part of any modern operating system can be implemented in several ways. Several methods have been in use for decades with each having their own implications and shortcomings. These have been tried to solve by the use of newer approaches with their implications and shortcomings. Therefore, improvement in scheduling algorithms being a chain of betterments. We expect this study to be helpful for further development of newer algorithms, which might enhance the efficiency of computers.

7. REFERENCES

[1] K. Venkata Manishankar, "A New Hybrid Scheduling Algorithm for Enhancement of CPU Performance", International Journal for Research in Applied Science and Engineering Technology, vol. 8, no. 6, pp. 2483-2490, 2020. Available: 10.22214/ijraset.2020.6400.

[2] "Real Time Systems - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/real-time-systems/.

[3] M. Nikravan and M. Kashani, "A genetic Algorithm for Process Scheduling in Distributed Operating Systems considering Load Balancing", 21st European Conference on Modelling and Simulation, 2007.

[4] K. Ramamritham and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems", Proceedings of the IEEE, vol. 82, no. 1, pp. 55-67, 1994. Available: 10.1109/5.259426.

[5] A. Negi and K. Pusukuri, "Applying Machine Learning Techniques to improve Linux Process Scheduling", TENCON 2005 2005 IEEE Region 10, 2005.

[6] J. Jose, O. Sujisha, M. Gilesh and T. Bindima, "On the Fairness of Linux O(1) Scheduler", International

Conference on Intelligent Systems Modelling and Simulations, pp. 668-674, 2014.

[7] N. Tahhan and M. Alasaady, "New Optimized Priority CPU Scheduling Algorithm by using Knapsack (NOPSACK)", Global Research and Development Journal for Engineering, vol. 5, no. 6, pp. 24-31, 2020.

[8] Neelakantagouda Patil,"A Knapsack Based CPU Process Scheduling Using Neelsack Algorithm", (IJSEAS) International Journal of Scientific Engineering and Applied Science, Volume-1, Issue-7, pp. 138-144, 2015.

[9] B. Nie, J. Du and G. Xu, "A New Operating System Scheduling Algorithm", Springer-Verlag Berlin Heidelberg 2011, pp. 92–96, 2011.