Experiment Name:

Design, Implementation, and Comparative Analysis of a Novel CPU Scheduling Algorithm with Existing Algorithms.

Objectives:

The objective of this project is to design, implement, and evaluate a novel CPU scheduling algorithm. Specifically, the project aims to:

1. Develop a new CPU scheduling algorithm to improve average waiting time, turnaround time, and CPU utilization
2. Implement the new algorithm and ensure it can handle various scheduling scenarios.
3. Compare the performance of the novel algorithm with existing algorithms such as FCFS, SJN, Priority Scheduling, and Round Robin using benchmark processes
4. Analyze the results to determine the effectiveness of the new algorithm and identify scenarios where it outperforms existing methods.
5. Document the design, implementation, and comparative analysis process, and provide a comprehensive report with data and observations.

Theory :

A CPU scheduling algorithm is a method used by an operating system to determine which processes in a system's ready queue are to allocate to the CPU for execution. The goal of these algorithms is to optimize various aspects of system performance, such as CPU utilization, throughput, turnaround time, waiting time, and response time.

Scheduling of work is done to finish the work on time. CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I/O etc., thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer. Whenever the CPU becomes idle, the operating system must select one of the processes in the line ready for launch. The selection process is done by a temporary (CPU) scheduler. The Scheduler selects between memory processes ready to launch and assigns the CPU to one of them.

Different CPU Scheduling algorithms have different structures and the choice of a particular algorithm depends on a variety of factors. Many conditions have been raised to compare CPU scheduling algorithms.

The criteria include the following:

- **CPU utilization:** The main purpose of any CPU algorithm is to keep the CPU as busy as possible. Theoretically, CPU usage can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the system load.
- **Throughput:** The average CPU performance is the number of processes performed and completed during each unit. This is called throughput. The output may vary depending on the length or duration of the processes.
- **Turn round Time:** For a particular process, the important conditions are how long it takes to perform that process. The time elapsed from the time of process delivery to the

time of completion is known as the conversion time. Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I /O.

- **Waiting Time:** The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.
- **Response Time:** In a collaborative system, turn around time is not the best option. The process may produce something early and continue to computing the new results while the previous results are released to the user. Therefore another method is the time taken in the submission of the application process until the first response is issued. This measure is called response time.

Types of CPU Scheduling Algorithms:

There are mainly two types of scheduling methods:

1. **Preemptive scheduling :** It is used when a process switches from running state to ready state or from the waiting state to the ready state.
2. **Non-Preemptive scheduling :** is used when a process terminates , or when a process switches from running state to waiting state.

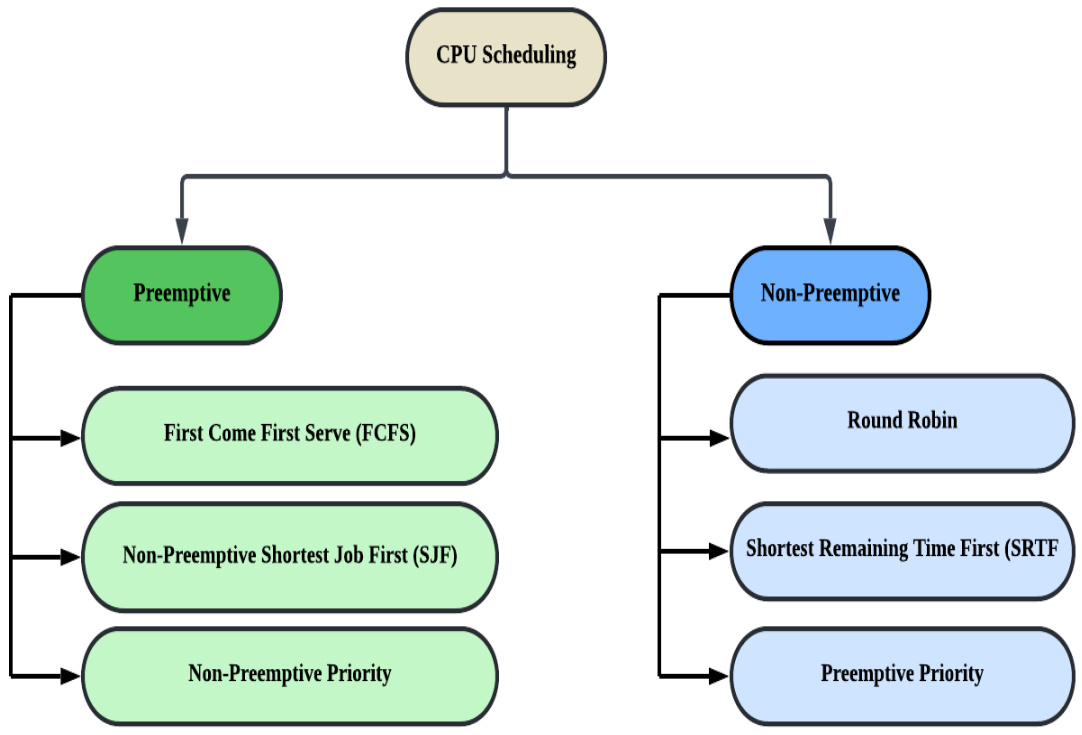

CPU Scheduling
Preemptive
First Come First Serve (FCFS)
Non-Preemptive Shortest Job First (SJF)
Non-Preemptive Priority
Non-Preemptive
Round Robin
Shortest Remaining Time First (SRTF
Preemptive Priority

Figure 1 : Types Of CPU Scheduling

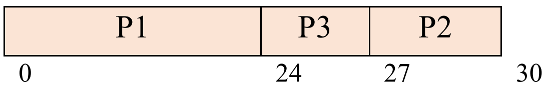
a) **First Come First Serve (FCFS) Scheduling Algorithm:** : According to this algorithm the process that arrives first in the ready queue is allocated to the CPU first. It is simple and easy to implement. FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, the process does not leave the CPU before completing its execution. This simplicity makes FCFS suitable for batch processing environments or scenarios where process arrival times are known in advance.

Example :

Data Table :

Process ID	Arrival Time	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	0	24	0	24	0	24
P2	3	3	24	30	24	27
P3	1	3	21	27	23	26

Gantt Chart:



Performance Table:

Average Response time	Average Turn Around time	Average Waiting Time
15	25.67	15.67

However, FCFS may lead to poor turnaround times, especially if long processes arrive first, causing shorter processes to wait for extended periods.

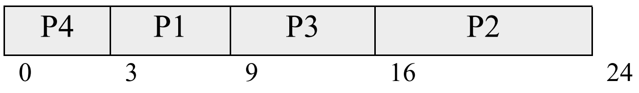
b) **Non-Preemptive Shortest Job First (SJF) Scheduling Algorithm:** This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Once a process is allocated to the CPU for execution it doesn't leave the CPU before completing its execution.

Example :

Data Table :

Process ID	Arrival Time	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	0	6	3	9	3	9
P2	0	8	16	24	16	24
P3	0	7	9	16	9	16
P4	0	3	0	3	0	3

Gantt Chart :



Performance Table:

Average Response time	Average Turn Around time	Average Waiting Time
7	13	7

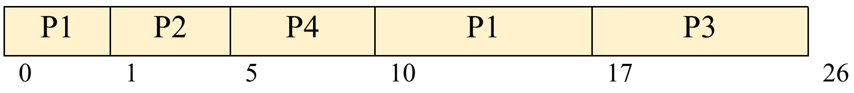
c) **Preemptive Shortest Job First (SJF) Scheduling Algorithm:** Shortest Remaining Time First (SRTF), also known as Preemptive Shortest Job First (SJF), is a CPU scheduling method that chooses the process with the least amount of CPU execution time left. A new process preempts an existing one if it arrives with less time left than the latter. This method prioritizes shorter activities, which reduces average waiting and turnaround times. But it can lead to frequent context flips, which can result in overhead, and it requires precise knowledge of burst durations. Fast response times are critical in real-time systems, where preemptive SJF is especially helpful.

Example :

Data Table :

Process ID	Arrival Time	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	0	8	0	17	9	17
P2	1	4	0	5	0	4
P3	2	9	15	26	15	24
P4	3	5	2	10	2	7

Gantt Chart :



Performance Table:

Average Response time	Average Turn Around time	Average Waiting Time
4.25	13	6.5

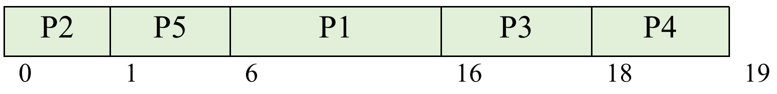
d) **Non-Preemptive Priority Scheduling Algorithm**: Non-preemptive Priority scheduling is a CPU scheduling algorithm that selects processes based on priority levels. The process with the highest priority receives the CPU and is allowed to finish before the next highest priority process is chosen. Higher priority is frequently indicated by lower numerical values. Although this approach guarantees that urgent activities are completed first, it may result in problems such as starvation, as low-priority operations might not be completed at all. A process that begins execution cannot be stopped by a new, higher-priority process until it completes its current burst, in contrast to preemptive algorithms.

Example :

Data Table :

Process ID	Arrival Time	Priority	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	0	3	10	6	16	6	16
P2	0	1	1	0	1	0	1
P3	0	4	2	16	18	16	18
P4	0	5	1	18	19	18	19
P5	0	2	5	1	6	1	6

Gantt Chart:



Performance Table:

Average Response time	Average Turn Around time	Average Waiting Time
8.2	12	8.2

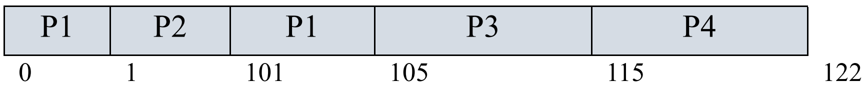
e) **Preemptive Priority Scheduling Algorithm :**This is a CPU scheduling algorithm that assigns priorities to processes and allocates the CPU to the process with the highest priority. Preemption allows the CPU to be taken away from a running process if a higher priority process arrives, ensuring that critical tasks are executed promptly. Scheduling decisions are based on priorities, with the highest priority process selected for execution. Priorities can be fixed or dynamically adjusted based on system conditions. Preemptive priority scheduling may lead to lower-priority tasks suffering from starvation as higher-priority tasks continuously arrive, causing unfair resource allocation. Additionally, priority inversion can occur when lower-priority tasks holding critical resources delay higher-priority tasks, impacting system performance. Managing priorities and preemption introduces complexity and overhead, potentially affecting system efficiency and stability.

Example :

Data Table :

Process ID	Arrival Time	Priority	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	0	3	10	6	16	6	16
P2	2	1	1	0	1	0	1
P3	3	4	2	16	18	16	18
P4	1	5	1	18	19	18	19
P5	4	2	5	1	6	1	6

Gantt Chart:



Performance Table:

Average Response time	Average Turn Around time	Average Waiting Time
6	11	7.2

f) **Round Robin Scheduling Algorithm:** Round Robin (RR) CPU scheduling algorithm is a widely utilized approach in multitasking environments. In RR scheduling, each process is allocated a fixed time slice, commonly known as a quantum. The scheduler cycles through the processes, providing each one with CPU time for the duration of its quantum. If a process fails to complete within its time slice, it is preempted and placed back in the ready queue to await its next turn. This method ensures fairness in CPU allocation as it grants each process an equal share of CPU time. Moreover, RR scheduling is relatively straightforward to implement and comprehend compared to other scheduling algorithms.

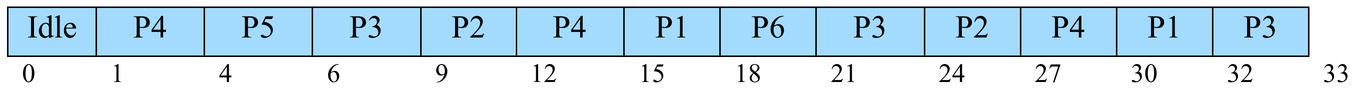
Example :

Data Table :

Process ID	Arrival Time	Burst Time	Response Time	Completion Time	Waiting Time	Turnaround time
P1	5	5	10	32	22	32
P2	4	6	6	27	17	27
P3	3	7	3	33	23	30
P4	1	9	0	30	20	29
P5	2	2	2	6	2	4
P6	6	3	3	21	12	15

Time Quantum is 3.

Gantt Chart:



Performance Table :

Average Response time	Average Turn Around time	Average Waiting Time
3.5	22.83	16

Round Robin (RR) CPU scheduling ensures fairness by providing each process with an equal opportunity for execution, preventing resource monopolization and catering well to multi-user systems. Its cyclic allocation of CPU time maintains low latency, crucial for interactive systems needing quick feedback. The simplicity of RR's implementation, involving ready queue management and fixed time quantum assignment, minimizes overhead. However, frequent context switching in RR can lead to high overhead, particularly with short time slices or numerous processes. It may not suit long-running or varied tasks, potentially reducing system performance. Additionally, improper time quantum settings can result in inefficient CPU utilization.

Our Proposed Algorithm:

A New Combination Approach to CPU Scheduling based on Priority and Round-Robin Algorithms for Assigning a Priority to a Process and Eliminating Starvation.

In our pursuit to enhance CPU scheduling, the primary objective of this approach was to address the issue of starvation in preemptive priority scheduling. By prioritizing processes based on their assigned priority and incorporating dynamic tie-breaking mechanisms, we aimed to ensure fairness and prevent certain processes from being indefinitely delayed due to higher-priority arrivals.

Our algorithm prioritizes fairness by incrementing the redundancy column for each executed process, ensuring that processes with fewer previous executions are favored when priorities are tied. Moreover, tie-breaking based on the shortest remaining burst time (SRBT) and first-come, first-served (FCFS) basis further reinforces fairness and efficient resource utilization.

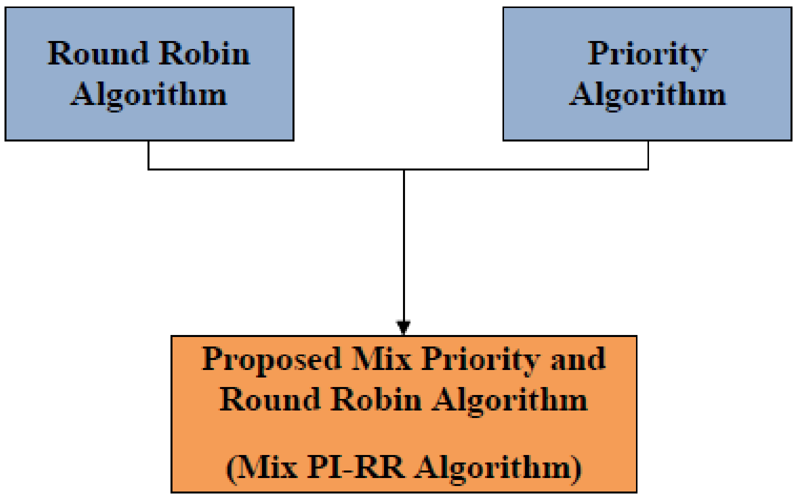


Figure 2 : Mix PI-RR Algorithm.

Additionally, the implementation of a dynamic time quantum calculation formula,

$$\textbf{Time Quantum}=\sqrt{\left(\frac{\textbf{sum of remaining burst time of the processes in ready queue}}{\textbf{number of the remaining processes in ready queue}}\right)}$$

This helps refine scheduling precision and mitigate the risk of starvation by ensuring that each process receives a fair share of CPU time. By combining these strategies, our approach aims to eliminate the occurrence of starvation in preemptive priority scheduling, thus promoting equitable process execution and overall system efficiency.

Features of the Proposed Algorithm:

1. Highest Priority Selection:

The CPU will select the process with the highest priority from the ready queue.

2. Redundancy Tracking:

When a process leaves the CPU, the redundancy column for that process (indicating the number of times a process has entered the CPU) is incremented.

3. Redundancy-Based Tie-Breaking:

If multiple processes have the same priority, the process with the fewest repetitions (lowest redundancy) is selected.

4. Shortest Remaining Burst Time (SRBT) Tie-Breaking:

If processes are tied in both priority and redundancy, the process with the shortest remaining burst time is selected.

5. **First-Come, First-Served (FCFS) Tie-Breaking:**

Even if ties persist, processes are executed on a first-come, first-served basis.

6. **Dynamic Priority Adjustment:**

After each three executions of a process, its priority is reassessed and lowered to the lowest priority value in the current ready queue. This mechanism prevents certain processes from monopolizing CPU resources and ensures fair access to processing time for all tasks.

7. **Preemptive Execution with Dynamic Time Quantum:**

Using the root mean of the remaining burst times of all processes in the ready queue for dynamic time quantum calculation offers a more balanced approach compared to using the root of square mean. The root mean considers all burst times equally, providing a fair representation of the current processing requirements of the tasks in the system.

When a process with a high burst time and high priority arrives initially, it indeed has the potential to monopolize the CPU if the time quantum calculation is not appropriately adjusted.

Here's why the root mean is a suitable choice:

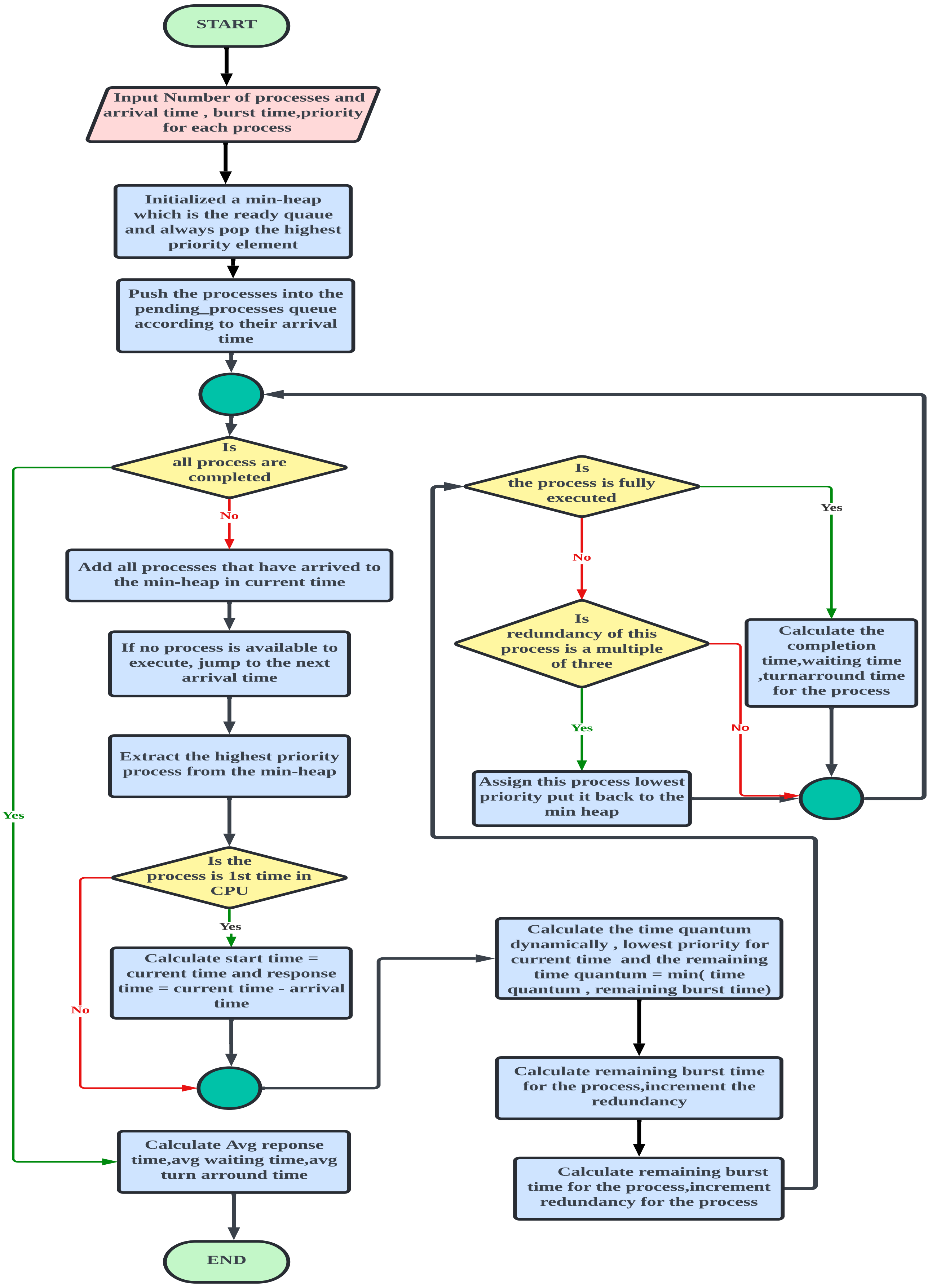
Equal Consideration: The root mean considers each burst time equally. It calculates the average by adding up all the burst times and dividing by the number of processes. This means that even if a few processes have significantly higher burst times, their impact on the calculated average is balanced by the burst times of other processes.

Mitigating Starvation: Using the root mean helps mitigate the risk of starvation by ensuring that processes with shorter burst times also receive a fair share of CPU time. If the time quantum were solely determined by processes with the highest burst times, shorter tasks might have to wait excessively long periods before receiving CPU time, leading to potential starvation.

Optimizing Scheduling Precision: By dynamically adjusting the time quantum based on the average burst time, the algorithm optimizes scheduling precision. Processes with longer burst times are allocated longer time slices, allowing them to progress efficiently, while shorter tasks are not unfairly penalized by excessively long time slices.

Overall, the root mean provides a balanced approach that considers the processing requirements of all tasks in the system, promoting fairness, mitigating starvation, and optimizing scheduling precision. This approach ensures that each process receives an appropriate share of CPU time, contributing to the overall efficiency and effectiveness of the scheduling algorithm.

Flow chart:



START
Input Number of processes and arrival time , burst time,priority for each process
Initialized a min-heap which is the ready quaue and always pop the highest priority element
Push the processes into the pending_processes queue according to their arrival time
Is all process are completed
No
Yes
Add all processes that have arrived to the min-heap in current time
If no process is available to execute, jump to the next arrival time
Extract the highest priority process from the min-heap
Is the process is 1st time in CPU
Yes
No
Calculate start time = current time and response time = current time - arrival time
Calculate Avg reponse time,avg waiting time,avg turn arround time
END
Is the process is fully executed
Yes
No
Is redundancy of this process is a multiple of three
Yes
No
Assign this process lowest priority put it back to the min heap
Calculate the completion time,waiting time ,turnarround time for the process
Calculate the time quantum dynamically , lowest priority for current time and the remaining time quantum = min(time quantum , remaining burst time)
Calculate remaining burst time for the process,increment the redundancy
Calculate remaining burst time for the process,increment redundancy for the process

Figure 3 : Flow-chart

Result:

Sample 1 :

Table 01 : Applied Preemptive Priority Algorithm

Process ID	Priority	Arrival Time	Brust time	Completion time	Response Time	Turn Around time	Waiting Time
0	2	0	5	105	0	105	100
1	1	1	100	101	0	100	0
2	3	2	10	115	103	113	103
3	4	3	7	122	112	119	112

Gantt Chart :

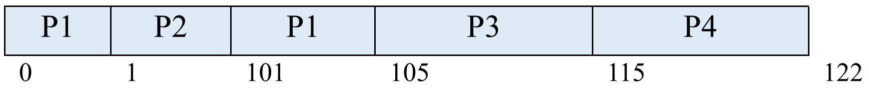
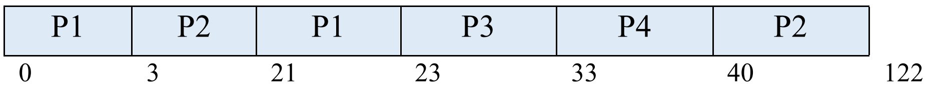


Table 02 : Applied Proposed Algorithm

Process ID	Priority	Arrival Time	Brust time	Completion time	Response Time	Turn Around time	Waiting Time
0	2	0	5	23	0	23	18
1	1	1	100	122	2	121	21
2	3	2	10	33	21	31	21
3	4	3	7	40	30	37	30

Gantt Chart :



Simulation of the proposed algorithm for this sample case :

🟦 Color in the ready queue represent the highest priority .

Step 01:

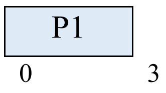
Current Time : 0 Unit

Calculated Time Quantum : 3

Ready Queue:

P1

Gantt Chart:



Step 02:

Current Time : 3 Unit

Calculated Time Quantum : 6

Ready Queue:

P2	P1	P3

Gantt Chart:

P1	P2

0 3 9

Step 03:

Current Time : 9 Unit

Calculated Time Quantum : 6

Ready Queue:

P2	P1	P3	P4

Gantt Chart:

P1	P2	P2

0 3 9 15

Step 04:

Current Time : 15 Unit

Calculated Time Quantum : 6 unit

Ready Queue:

P2	P1	P3	P4

Gantt Chart:

P1	P2	P2	P2

0 3 9 15 21

Step 05:

Current Time : 21 Unit

Calculated Time Quantum : 6

Ready Queue:

P1	P3	P4	P2

Gantt Chart:

P1	P2	P2	P2	P1

0 3 9 15 21 23

<u>Step 06:</u>

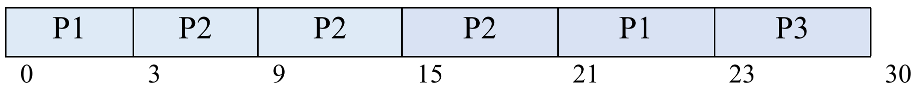
Time : 23 Unit

Calculated Time Quantum : 7

Ready Queue:

P3	P4	P2

Gantt Chart:



<u>Step 07:</u>

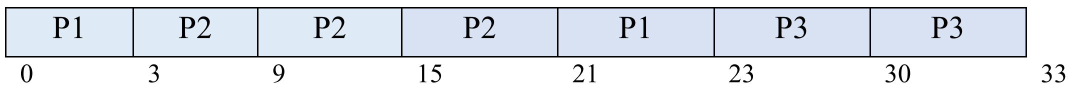
Time : 30 Unit

Calculated Time Quantum : 6

Ready Queue:

P3	P4	P2

Gantt Chart:



<u>Step 08:</u>

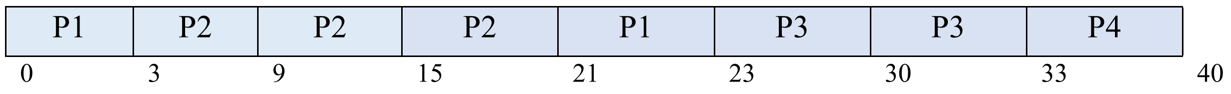
Time : 33 Unit

Calculated Time Quantum : 7

Ready Queue:

P4	P2

Gantt Chart:



<u>Step 09:</u>

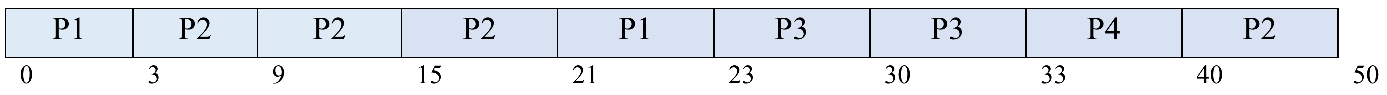
Time : 40 Unit

Calculated Time Quantum : 10

Ready Queue:

P2

Gantt Chart:

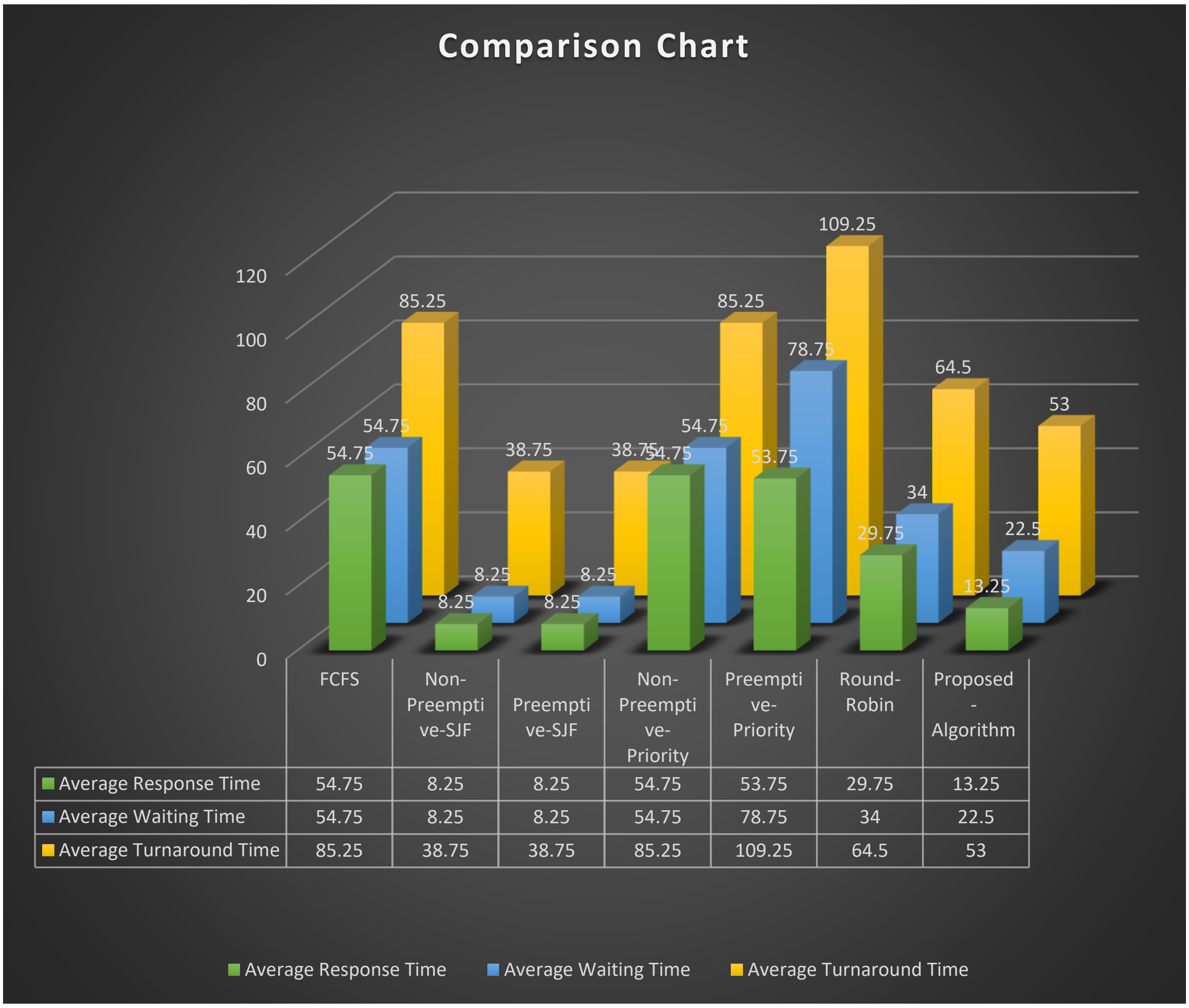


Now only P2 is left. After few steps we will get the final gantt chart.

Final Gantt Chart :

P1	P2	P1	P3	P4	P2	
0	3	21	23	33	40	122

Compare All Scheduling Algorithms For The Sample Case :



	FCFS	Non-Preemptive-SJF	Preemptive-SJF	Non-Preemptive-Priority	Preemptive-Priority	Round-Robin	Proposed-Algorithm
🟩 Average Response Time	54.75	8.25	8.25	54.75	53.75	29.75	13.25
🟦 Average Waiting Time	54.75	8.25	8.25	54.75	78.75	34	22.5
🟨 Average Turnaround Time	85.25	38.75	38.75	85.25	109.25	64.5	53

Figure 4 : Comparison chart

Drawbacks of the Proposed Algorithm:

While the proposed CPU scheduling algorithm enhances fairness and efficiency through dynamic time quantum calculation and redundancy-based priority adjustments, it has certain limitations. The increased computational complexity and frequent context switching can lead to higher overhead compared to simpler algorithms. This complexity might also make the algorithm challenging to implement and maintain. Additionally, processes with long burst times could still dominate CPU time, potentially delaying shorter processes. In scenarios with many high-priority arrivals, lower-priority processes might face delays, leading to potential starvation. The algorithm may not be suitable for real-time systems due to its unpredictability and resource demands. Overall, while it offers significant benefits, the algorithm may not be ideal for all cases, particularly those requiring minimal overhead and high predictability.

Conclusion:

In this study, we designed, implemented, and evaluated a novel CPU scheduling algorithm aimed at addressing the issue of starvation in preemptive priority scheduling. Our algorithm integrates several innovative features, including dynamic priority handling, redundancy tracking, and a dynamic time quantum calculation based on the root mean of remaining burst times.

The proposed algorithm provides a robust solution to the challenge of starvation in preemptive priority scheduling, promoting equitable process execution and overall system efficiency. This advancement has significant implications for real-world applications, where fair and efficient CPU scheduling is critical for system performance and user satisfaction.

In conclusion, our novel CPU scheduling algorithm successfully addresses the long-standing issue of starvation in preemptive priority scheduling, offering a balanced and efficient approach to process management. This study lays the groundwork for further innovations in CPU scheduling, contributing to the ongoing advancement of operating system performance and fairness.