**Interactive Learning Showcase Report Project**

**Title: Preemptive scheduling-Priority Scheduling**

**Date**: 07-10-2025

**Team Members:**

**HARISHWER B N(RA2411029010060)  
ABISHEK E(RA2411029010045)  
UDAY KUMAR(RA2411029010052)**

**Abstract**

Priority scheduling is a widely used CPU scheduling algorithm in operating systems, allowing processes to run based on priority levels. It supports both preemptive and non-preemptive variants, balancing prompt attention to critical tasks with the risk of starvation for lower-priority processes. Techniques like aging help address these drawbacks, ensuring fair CPU allocation.

**Introduction**

An operating system coordinates CPU usage among multiple processes using scheduling algorithms. Priority scheduling assigns priority values to each process, dictating execution order. It is particularly effective in situations where certain tasks must receive preferential treatment, such as real-time or system-critical processes.

**Priority Scheduling: Concept and Types**

**Concept:**

* Each process is assigned a priority; those with higher priority (often represented by a *lower* number) are scheduled before others.
* Processes with equal priority are usually scheduled using another method, such as First-Come-First-Served (FCFS).

**Types:**

1. **Preemptive Priority Scheduling:**
   * A process can be interrupted if a new or waiting process with a higher priority becomes ready, preempting the CPU.
2. **Non-Preemptive Priority Scheduling:**
   * Once a process starts, it runs to completion or until it voluntarily yields the CPU. Lower-priority processes wait until higher-priority ones finish.

**Key Properties: Starvation and Aging**

* **Starvation:** Occurs when low-priority processes wait indefinitely because higher-priority processes continue to arrive, blocking CPU access.
* **Aging:** A technique to prevent starvation, it gradually increases the priority of waiting processes over time, ensuring that every process will eventually be executed.

**Description of the Concept**

Priority scheduling can be implemented in both preemptive and non-preemptive modes. The scheduler always selects the highest-priority ready process to run next. Priority values may be set internally (based on resource needs, memory usage, etc.) or externally (user, importance level). Applications that require guaranteed response times, like real-time control systems, often use this algorithm.

**Formulas Used**

Let:

* = Arrival Time
* = Burst Time
* = Completion Time
* = Turnaround Time
* = Waiting Time

Formulas:

* **Average Turnaround Time:**
* **Average Waiting Time:**

**Sample Code**

Here’s a simple C++ code for non-preemptive priority scheduling:

cpp

#**include**<bits/stdc++.h>

**using** **namespace** std;

**struct** Process {

**int** pid; *// Process ID*

**int** bt; *// CPU Burst time required*

**int** priority;

};

**bool** compare(Process a, Process b) {

**return** (a.priority > b.priority); *// Higher value means higher priority*

}

**void** waitingtime(Process pro[], **int** n, **int** wt[]) {

wt[0] = 0;

**for** (**int** i = 1; i < n ; i++)

wt[i] = pro[i-1].bt + wt[i-1];

}

**void** turnarround(Process pro[], **int** n, **int** wt[], **int** tat[]) {

**for** (**int** i = 0; i < n ; i++)

tat[i] = pro[i].bt + wt[i];

}

**void** avgtime(Process pro[], **int** n) {

**int** wt[n], tat[n], total\_wt = 0, total\_tat = 0;

waitingtime(pro, n, wt);

turnarround(pro, n, wt, tat);

cout << "Processes Burst time Waiting time Turn around time\n";

**for** (**int** i = 0; i < n; i++) {

total\_wt += wt[i];

total\_tat += tat[i];

cout << pro[i].pid << "\t\t" << pro[i].bt << "\t\t" << wt[i] << "\t\t" << tat[i] << endl;

}

cout << "\nAverage waiting time = " << (**float**)total\_wt / (**float**)n;

cout << "\nAverage turn around time = " << (**float**)total\_tat / (**float**)n;

}

**void** scheduling(Process pro[], **int** n) {

sort(pro, pro + n, compare);

cout << "Order in which processes gets executed \n";

**for** (**int** i = 0 ; i < n; i++)

cout << pro[i].pid << " ";

cout << endl;

avgtime(pro, n);

}

**int** main() {

Process pro[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};

**int** n = **sizeof** pro / **sizeof** pro[0];

scheduling(pro, n);

**return** 0;

}

**Advantages and Disadvantages**

**Advantages:**

* Crucial tasks get prompt attention
* Suitable for real-time and critical systems
* Can improve average waiting time for high-priority processes

**Disadvantages:**

* Can cause starvation of low-priority processes
* May require extra logic to handle starvation (e.g., aging)
* Priority inversion can occur if not managed correctly
* Overhead in maintaining and updating priorities

**Applications**

* Real-time systems
* Embedded controllers
* Operating Systems for desktops and servers
* Print spooling and traffic management
* Environments requiring rapid response for high-priority jobs

**Preventing Starvation and Use of Aging**

Starvation can occur if high-priority processes keep arriving, blocking lower-priority processes forever. **Aging** is a common technique to prevent this:

* Gradually increase the priority of waiting processes over time.
* Ensures every process eventually becomes the highest priority if it waits long enough.

Example: For every time unit a process waits, its priority is increased by 1. Over time, even low-priority processes will be executed.

**Conclusion**

Priority scheduling is a flexible and efficient algorithm for CPU management in operating systems, balancing the need to serve critical tasks quickly while maintaining overall fairness. While it presents risks like starvation, practical solutions such as aging ensure fair access for all processes. Priority scheduling remains vital for systems requiring differentiated process handling.