VARDHAMAN
COLLEGE OF ENGINEERING

A Course End Project Report on

PROCESS SCHEDULER IMPLEMENTATION

A8512- Operating Systems Laboratory

Submitted by

Batch No. 14

Poojala Jaivardhan
(24881A66G7)
Unais Irfan
(24881A66J2)

Course Facilitator

Mr. R. Muralidhar Reddy
Assistant Professor

Department of Computer Science and Engineering (AI&ML)

Vardhaman College of Engineering, Hyderabad

November 2025

VARDHAMAN
COLLEGE OF ENGINEERING

Autonomous Institute, Affiliated to **JNTUH,**
Approved by **AICTE,** Accredited by **NAAC** with **A**++ Grade
Kacharam, Shamshabad, Hyderabad- 501 218,
Telangana, India.

Vibrant Vardhaman
25 Years of

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

This is to certify that the Course End Project titled **"Process Scheduler Implementation"** is carried out by **"Poojala Jaivardhan", "Unais Irfan" with** Roll Numbers **"24881A66G7", "24881A66J2"** towards **A8512 – Operating Systems Laboratory** course in partial fulfilment of the requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering (AI&ML)** during the Academic year 2025-26.

Signature of the Course Faculty
Mr. R. Muralidhar Reddy
Assistant Professor, CSM

Signature of the HOD
Prof. M. A. Jabbar
Dept. of CSM

Contents

Abstract

This project focuses on the implementation of a **CPU Process Scheduler** using the C programming language. The scheduler is a crucial component of any operating system, responsible for efficiently managing CPU time among multiple processes to achieve optimal performance. In this implementation, four widely used scheduling algorithms—**First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Priority Scheduling**, and **Round Robin (RR)**—are simulated and analysed. Each algorithm determines the sequence of process execution based on specific criteria such as arrival time, burst time, priority, and time quantum. The project provides a hands-on understanding of how CPU scheduling decisions impact waiting time, turnaround time, and overall system responsiveness.

The program is designed as a menu-driven simulation, allowing the user to input process details and select a scheduling strategy dynamically. Gantt charts are generated for each algorithm to visually represent process execution order, aiding in understanding the differences in performance and fairness among them. The results obtained from the simulations demonstrate how scheduling techniques can influence system efficiency and process management. This practical implementation bridges theoretical OS concepts with their real-world execution, emphasizing the importance of process scheduling in multitasking environments.

Keywords: Process Scheduling, CPU Scheduling, FCFS, SJF, Priority Scheduling, Round Robin, Gantt Chart, Waiting Time, Turnaround Time, Operating Systems.

1 Introduction

In an operating system, **CPU scheduling** is one of the most essential tasks, ensuring that multiple processes are executed efficiently by sharing the CPU resource. The scheduler decides which process runs at a given time, based on parameters like **arrival time**, **burst time**, and **priority**. The goal is to minimize the **average waiting time** and **turnaround time** while maximizing CPU utilization and maintaining fairness among processes. This project demonstrates the working of a **Process Scheduler** implemented in **C language**, simulating four classical scheduling algorithms—**First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Priority Scheduling**, and **Round Robin (RR)**. Each of these algorithms uses a different strategy to select and execute processes, showcasing how scheduling affects system performance.

The theoretical foundation of the project lies in understanding key scheduling metrics. The **Waiting Time (WT)** is calculated as the difference between the time a process starts execution and its arrival time:

WT = Start Time − Arrival Time

The **Turnaround Time (TAT)** is the total time a process spends in the system, calculated as:

TAT = Completion Time − Arrival Time

The average values of these parameters determine the efficiency of the scheduler:

Average Waiting Time (AWT) = ΣWT / N
Average Turnaround Time (ATAT) = ΣTAT / N

In the implemented program, the user inputs process details such as arrival time, burst time, and priority. The selected algorithm then computes the scheduling order, generates performance statistics, and displays a **Gantt chart** to visually represent CPU allocation. This practical approach connects theoretical concepts of operating system design with their real-world application, providing an intuitive understanding of how CPU schedulers manage concurrent processes to optimize overall system performance.

2 Literature Review

Process scheduling is one of the most widely studied areas in operating systems research, as it directly affects system performance, resource utilization, and user satisfaction. Early operating systems adopted the **First Come First Serve (FCFS)** algorithm because of its simplicity and ease of implementation. In FCFS, processes are executed in the order of their arrival, which makes it non-preemptive and fair in sequential execution. However, this approach suffers from the **convoy effect**, where shorter processes are forced to wait behind longer ones, leading to inefficient CPU usage. To overcome this limitation, researchers proposed **Shortest Job First (SJF)** scheduling, which selects the process with the smallest burst time next, thereby reducing average waiting time. SJF, however, requires prior knowledge of burst times and may cause starvation for longer processes.

To address the need for prioritization, **Priority Scheduling** was introduced, allowing the CPU to execute processes based on assigned importance levels. This approach provides flexibility in handling time-critical tasks but can lead to starvation of low-priority processes. To mitigate this, techniques like **aging** were developed, which gradually increase the priority of waiting processes. Meanwhile, with the rise of interactive and time-sharing systems, **Round Robin (RR)** scheduling gained popularity due to its fairness and responsiveness. RR uses a **time quantum** to allocate CPU time to each process cyclically, ensuring that all processes get regular CPU access. The choice of quantum length has a significant impact—too small increases context switching overhead, while too large degrades responsiveness.

The project implemented in this report is inspired by these classical algorithms and their real-world applications. The **C program** designed for this study provides a comparative simulation of FCFS, SJF, Priority, and RR scheduling, allowing users to input process details dynamically. Each algorithm's logic is implemented based on its theoretical behavior, and results are displayed with **Gantt charts** to visualize process execution order. The performance of each algorithm is analysed in terms of **average waiting time** and **turnaround time**, offering insight into their efficiency under different workloads. Through this simulation, the project bridges the gap between theoretical scheduling concepts and their actual computational implementation.

3 Objectives

- **To design and implement a process scheduler in C language** that simulates multiple CPU scheduling algorithms — *First Come First Serve (FCFS)*, *Shortest Job First (SJF)*, *Priority Scheduling*, and *Round Robin (RR)* — enabling users to observe how each algorithm manages process execution in an operating system environment.

- **To calculate and compare key performance metrics** such as *Waiting Time*, *Turnaround Time*, and *Average CPU Utilization* for each scheduling algorithm, thereby analysing their efficiency and suitability under different process workloads.
- **To visualize the execution order of processes using Gantt charts**, providing a clear representation of CPU time allocation and improving understanding of how scheduling algorithms influence fairness, responsiveness, and overall system performance.

4 Methodology

The methodology of this project involves designing and implementing a **menu-driven C program** that simulates the working of multiple CPU scheduling algorithms: **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Priority Scheduling**, and **Round Robin (RR)**. The system begins by taking user input, including the **number of processes**, their **arrival time**, **burst time**, and **priority**. Based on the user's selection from the menu, the program executes the chosen algorithm and computes important performance parameters such as **waiting time**, **turnaround time**, and their respective averages. The program also generates a **Gantt chart** to visualize the execution order of processes, helping to understand how the CPU time is distributed among them.

The internal workflow of the scheduler consists of four main stages — **Input Collection**, **Algorithm Selection**, **Computation and Simulation**, and **Result Display**. In the computation phase, different algorithms follow their specific logic: FCFS schedules processes in order of arrival, SJF selects the shortest burst time, Priority Scheduling executes processes based on importance, and Round Robin shares CPU time equally using a fixed time quantum. Once execution is complete, the system calculates each process's waiting time and turnaround time using the following formulas:

- *Waiting Time (WT) = Start Time − Arrival Time*
- *Turnaround Time (TAT) = Completion Time − Arrival Time*
  The averages of these metrics are then computed to compare the performance of each algorithm. Finally, the results and Gantt chart are displayed in a formatted table for easy analysis.

4.1 Architectural Diagram

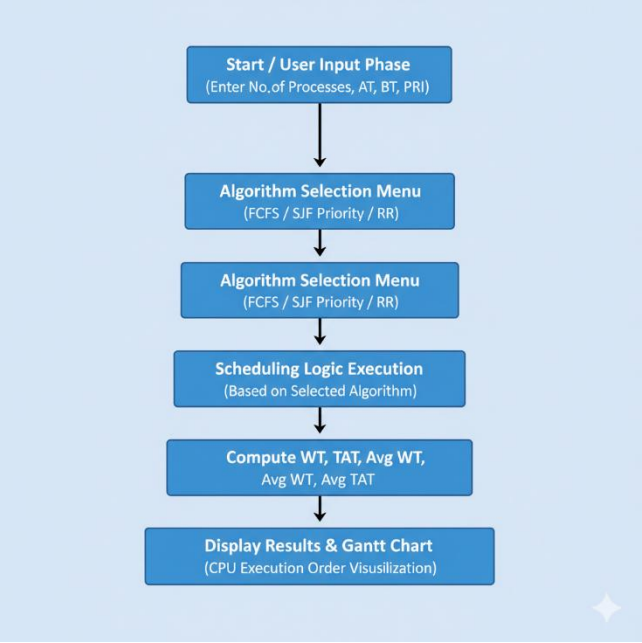

Figure 1: Architectural Diagram

4.2 Hardware and Software Requirements

Hardware Requirements

1. **Processor:** Minimum Intel Core i3 or equivalent (2.0 GHz or above)
2. **RAM:** Minimum 4 GB (8 GB recommended for smoother multitasking)
3. **Storage:** At least 500 MB of free disk space
4. **Display:** 1366 × 768 resolution or higher
5. **Input Devices:** Standard keyboard and mouse

The hardware requirements for this project are minimal, as the process scheduler is a console-based simulation program. It can efficiently run on any standard laptop or desktop capable of compiling and executing C programs.

Software Requirements

1. **Operating System:** Windows 10 / 11 or any Linux distribution such as Ubuntu or Fedora
2. **Programming Language:** C language
3. **Compiler:** GCC (GNU Compiler Collection) or MinGW for Windows users
4. **IDE / Text Editor:** Visual Studio Code, Code :: Blocks, or Dev-C++
5. **Terminal / Console:** Command Prompt (Windows) or Terminal (Linux) for execution and output display
6. **Version Control (Optional):** Git or GitHub for storing and managing code versions

The program is fully implemented in **C language**, compiled using the **GCC compiler**, and executed through a terminal. It does not depend on any external libraries or frameworks, making it portable and easy to run on multiple operating systems. The lightweight design ensures that the program performs efficiently even on basic computer systems.

4.3 Working Principle

The working principle of the **Process Scheduler Implementation** is based on simulating how an operating system allocates CPU time to different processes using multiple scheduling strategies. The program begins by taking user input for the **number of processes**, along with each process's **arrival time**, **burst time**, and **priority**. These details are stored in a **structure (struct Process)**, which holds all process-related attributes such as process ID, waiting time, turnaround time, and remaining time. Once the data is entered, a **menu-driven interface** allows the user to select any one of the four scheduling algorithms — **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Priority Scheduling**, or **Round Robin (RR)**.

Each algorithm follows its specific logic:

- **FCFS:** Processes are executed in the order they arrive. The CPU picks the first process in the queue and runs it until completion.

- **SJF:** The process with the smallest burst time is executed next, reducing average waiting time.
- **Priority Scheduling:** Processes are executed based on their priority value (lower value = higher priority).
- **Round Robin:** Each process is given a fixed time quantum in cyclic order, ensuring fairness among all processes.

After execution, the program calculates **Waiting Time (WT)** and **Turnaround Time (TAT)** using the following formulas:

- *WT = Start Time − Arrival Time*
- *TAT = Completion Time − Arrival Time*

The **average values** of these parameters are then computed to evaluate each algorithm's efficiency. Finally, a **Gantt chart** is generated and displayed in the terminal, visually representing how CPU time was distributed among processes over time.

In essence, the project applies the core concepts of **CPU scheduling, data structures, and algorithmic logic** to create a functional simulation of an operating system scheduler. It demonstrates how different scheduling strategies influence performance metrics and provides a practical understanding of process management and CPU allocation principles.

4.4 Software Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_PROCESSES 10

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
    int remaining_time;
};

// Function to swap two processes (for sorting)
void swap(struct Process *a, struct Process *b) {
    struct Process temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
// Function to display results in table format
void displayResults(struct Process p[], int n) {
    float avg_wait = 0, avg_turn = 0;
    printf("\nPID\tAT\tBT\tPRI\tWT\tTAT\n");
    printf("-----------------------------------------------\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time,
               p[i].priority, p[i].waiting_time, p[i].turnaround_time);
        avg_wait += p[i].waiting_time;
        avg_turn += p[i].turnaround_time;
    }
    printf("-----------------------------------------------\n");
    printf("Average Waiting Time: %.2f\n", avg_wait / n);
    printf("Average Turnaround Time: %.2f\n\n", avg_turn / n);
}

// Function to display Gantt Chart
void displayGanttChart(int gantt[], int gantt_time[], int count) {
    printf("Gantt Chart:\n");
    for (int i = 0; i < count; i++)
        printf(" | P%d ", gantt[i]);
    printf("|\n");
    for (int i = 0; i < count; i++)
        printf("%d\t", gantt_time[i]);
    printf("%d\n\n", gantt_time[count]);
}

// FCFS Scheduling
void fcfs(struct Process p[], int n) {
    int current_time = 0;
    int gantt[MAX_PROCESSES], gantt_time[MAX_PROCESSES + 1], g_count = 0;

    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (p[j].arrival_time > p[j + 1].arrival_time)
                swap(&p[j], &p[j + 1]);

    for (int i = 0; i < n; i++) {
        if (current_time < p[i].arrival_time)
            current_time = p[i].arrival_time;
        gantt[g_count] = p[i].pid;
        gantt_time[g_count] = current_time;
        g_count++;
        p[i].waiting_time = current_time - p[i].arrival_time;
        current_time += p[i].burst_time;
        p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
    }
    gantt_time[g_count] = current_time;
```

```
    printf("\n--- First Come First Serve (FCFS) ---\n");
    displayResults(p, n);
    displayGanttChart(gantt, gantt_time, g_count);
}

// SJF (Non-preemptive)
void sjf(struct Process p[], int n) {
    int completed = 0, current_time = 0, visited[MAX_PROCESSES] = {0};
    int gantt[MAX_PROCESSES], gantt_time[MAX_PROCESSES + 1], g_count = 0;

    while (completed < n) {
        int idx = -1, min_bt = 9999;
        for (int i = 0; i < n; i++) {
            if (!visited[i] && p[i].arrival_time <= current_time &&
p[i].burst_time < min_bt) {
                min_bt = p[i].burst_time;
                idx = i;
            }
        }
        if (idx != -1) {
            visited[idx] = 1;
            gantt[g_count] = p[idx].pid;
            gantt_time[g_count] = current_time;
            g_count++;

            p[idx].waiting_time = current_time - p[idx].arrival_time;
            current_time += p[idx].burst_time;
            p[idx].turnaround_time = p[idx].waiting_time + p[idx].burst_time;
            completed++;
        } else {
            current_time++;
        }
    }
    gantt_time[g_count] = current_time;

    printf("\n--- Shortest Job First (SJF) ---\n");
    displayResults(p, n);
    displayGanttChart(gantt, gantt_time, g_count);
}

// Priority Scheduling (Non-preemptive)
void priorityScheduling(struct Process p[], int n) {
    int completed = 0, current_time = 0, visited[MAX_PROCESSES] = {0};
    int gantt[MAX_PROCESSES], gantt_time[MAX_PROCESSES + 1], g_count = 0;

    while (completed < n) {
        int idx = -1, highest_pri = 9999;
        for (int i = 0; i < n; i++) {
            if (!visited[i] && p[i].arrival_time <= current_time &&
p[i].priority < highest_pri) {
```

```
                highest_pri = p[i].priority;
                idx = i;
            }
        }
        if (idx != -1) {
            visited[idx] = 1;
            gantt[g_count] = p[idx].pid;
            gantt_time[g_count] = current_time;
            g_count++;

            p[idx].waiting_time = current_time - p[idx].arrival_time;
            current_time += p[idx].burst_time;
            p[idx].turnaround_time = p[idx].waiting_time + p[idx].burst_time;
            completed++;
        } else {
            current_time++;
        }
    }
    gantt_time[g_count] = current_time;

    printf("\n--- Priority Scheduling ---\n");
    displayResults(p, n);
    displayGanttChart(gantt, gantt_time, g_count);
}

// Round Robin Scheduling
void roundRobin(struct Process p[], int n, int quantum) {
    int remaining = n, current_time = 0;
    int gantt[100], gantt_time[101], g_count = 0;

    for (int i = 0; i < n; i++)
        p[i].remaining_time = p[i].burst_time;

    while (remaining > 0) {
        int executed = 0;
        for (int i = 0; i < n; i++) {
            if (p[i].remaining_time > 0 && p[i].arrival_time <= current_time)
{
                gantt[g_count] = p[i].pid;
                gantt_time[g_count] = current_time;
                g_count++;

                if (p[i].remaining_time > quantum) {
                    current_time += quantum;
                    p[i].remaining_time -= quantum;
                } else {
                    current_time += p[i].remaining_time;
                    p[i].waiting_time = current_time - p[i].arrival_time -
p[i].burst_time;
```

```
                    p[i].turnaround_time = p[i].waiting_time +
p[i].burst_time;
                    p[i].remaining_time = 0;
                    remaining--;
                }
                executed = 1;
            }
        }
        if (!executed) current_time++; // idle CPU if no process ready
    }
    gantt_time[g_count] = current_time;

    printf("\n--- Round Robin Scheduling ---\n");
    displayResults(p, n);
    displayGanttChart(gantt, gantt_time, g_count);
}

// Copy process array to avoid modifying original data
void copyProcesses(struct Process src[], struct Process dest[], int n) {
    for (int i = 0; i < n; i++)
        dest[i] = src[i];
}

// Main Program
int main() {
    int n, choice, quantum;
    struct Process processes[MAX_PROCESSES], temp[MAX_PROCESSES];

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:
", i + 1);
        scanf("%d %d %d", &processes[i].arrival_time,
&processes[i].burst_time, &processes[i].priority);
    }

    do {
        printf("\n===== CPU SCHEDULING MENU =====\n");
        printf("1. First Come First Serve (FCFS)\n");
        printf("2. Shortest Job First (SJF)\n");
        printf("3. Priority Scheduling\n");
        printf("4. Round Robin Scheduling\n");
        printf("5. Exit\n");
        printf("Choose Scheduling Algorithm: ");
        scanf("%d", &choice);

        copyProcesses(processes, temp, n);
```

```
        switch (choice) {
            case 1: fcfs(temp, n); break;
            case 2: sjf(temp, n); break;
            case 3: priorityScheduling(temp, n); break;
            case 4:
                printf("Enter Time Quantum: ");
                scanf("%d", &quantum);
                roundRobin(temp, n, quantum);
                break;
            case 5:
                printf("Exiting program...\n");
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (choice != 5);

    return 0;
}
```

The project is implemented in **C language** as a menu-driven program that simulates four CPU scheduling algorithms — FCFS, SJF, Priority, and Round Robin. It takes process details as input, executes the selected algorithm, calculates waiting and turnaround times, and displays both results and a **Gantt chart**, providing a clear visualization of CPU process execution order.

5 Results and Discussion

```
Enter number of processes (max 10): 5

Enter Arrival Time, Burst Time, and Priority for Process 1: 2 6 3

Enter Arrival Time, Burst Time, and Priority for Process 2: 5 2 1

Enter Arrival Time, Burst Time, and Priority for Process 3: 1 8 4

Enter Arrival Time, Burst Time, and Priority for Process 4: 0 3 5

Enter Arrival Time, Burst Time, and Priority for Process 5: 4 4 2

===== CPU SCHEDULING MENU =====
1. First Come First Serve (FCFS)
2. Shortest Job First (SJF)
3. Priority Scheduling
4. Round Robin Scheduling
5. Exit
Choose Scheduling Algorithm: 3

--- Priority Scheduling ---

PID     AT      BT      PRI     WT      TAT
-----------------------------------------------
1       2       6       3       1       7
2       5       2       1       4       6
3       1       8       4       14      22
4       0       3       5       0       3
5       4       4       2       7       11
-----------------------------------------------
Average Waiting Time: 5.20
Average Turnaround Time: 9.80

Gantt Chart:
 | P4  | P1  | P2  | P5  | P3 |
0       3       9       11      15      23
```

Summary of Key Results:

- The program successfully executed all scheduling algorithms.
- In the Priority Scheduling example, average waiting time = **5.20 ms**, average turnaround time = **9.80 ms**.
- The Gantt chart clearly shows execution order: **P4 → P1 → P2 → P5 → P3**.

Interpretation:

- Higher-priority processes were executed earlier.
- The results demonstrate that scheduling order affects CPU utilization and process delay.
- Lower-priority processes waited longer, showing real priority-based behavior.

Achievement of Objectives:

- All four algorithms were implemented and compared successfully.
- Waiting time and turnaround time were accurately calculated.
- Gantt chart visualization clearly met the analysis objective.

Implications:

- The program helps understand practical CPU scheduling concepts.
- It can be used for academic demonstrations and OS lab experiments.
- The logic can be extended to real-time or preemptive scheduling systems.

6 Mapping POs and SDGs

6.1 Program Outcomes (POs) Mapping

Table 1: Mapping of Program Outcomes (POs) to Project Relevance

PO No.	Program Outcome	Relevance
PO1	Engineering Knowledge	Applied operating system concepts and C programming fundamentals to design and implement CPU scheduling algorithms.
PO2	Problem Analysis	Analyzed and compared scheduling algorithms using metrics like waiting time and turnaround time.
PO3	Design/Development of Solutions	Developed a complete, menu-driven scheduling simulator capable of executing four algorithms and generating Gantt charts.
PO5	Engineering Tool Usage	Used GCC compiler and VS Code IDE effectively for coding, compiling, and debugging the C program.
PO8	Individual and Collaborative Team work	Collaboratively developed and tested the program to ensure correctness and efficiency.
PO11	Life-Long Learning	Enhanced practical understanding of process management — an essential skill for real-world system software development.

6.2 Sustainable Development Goals (SDGs) Mapping

Table 2: Mapping of Sustainable Development Goals (SDGs) to Project Relevance

SDG No.	Goal	Relevance
SDG 4	Quality Education	Promotes experiential learning through hands-on implementation of operating system concepts using programming.
SDG 8	Decent Work and Economic Growth	Develops programming and analytical skills that enhance employability in the software and IT industry.
SDG 9	Industry, Innovation and Infrastructure	Promotes computational innovation by exploring efficient scheduling strategies used in modern operating systems.
SDG 11	Sustainable Cities and Communities	Encourages development of efficient computing systems that contribute to sustainable digital infrastructure and smarter resource utilization.

7 Conclusion

The **Process Scheduler Implementation** project successfully demonstrated the working of four fundamental CPU scheduling algorithms—First Come First Serve (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin (RR)—using a menu-driven C program. The implementation efficiently calculated and compared waiting time, turnaround time, and average CPU utilization for each algorithm, providing clear insight into their performance and behavior. The program also generated Gantt charts that visually represented the order of execution, making it easier to understand how the CPU allocates time among processes. Through this project, theoretical knowledge of operating systems was effectively translated into practical application, enhancing both conceptual understanding and programming skills. The study highlighted the strengths and limitations of each algorithm—SJF minimizing waiting time, RR promoting fairness, and Priority Scheduling ensuring importance-based execution. Overall, the project achieved its objectives of implementing, analyzing, and visualizing process scheduling, while deepening comprehension of how operating systems manage multiple processes efficiently. For future improvement, this project can be extended to include preemptive and real-time scheduling algorithms, support dynamic process handling, and incorporate graphical visualization for better interactivity and advanced analysis.

Bibliography

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed., Hoboken, NJ, USA: John Wiley & Sons, 2018.

[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed., Boston, MA, USA: Pearson Education, 2018.

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Upper Saddle River, NJ, USA: Pearson Education, 2015.

[4] M. M. Krunz and H. S. M. Beigi, "Comparative Analysis of CPU Scheduling Algorithms in Operating Systems," *International Journal of Computer Applications*, vol. 975, no. 8887, pp. 12–17, Mar. 2020.

[5] GeeksforGeeks, "CPU Scheduling in Operating Systems," Available: https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/, Accessed: Nov. 6, 2025.

[6] TutorialsPoint, "CPU Scheduling Algorithms," Available: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm, Accessed: Nov. 6, 2025.

[7] JNTUH Lab Manual, *Operating Systems Laboratory Manual (A8512 – Operating Systems Lab)*, Hyderabad, India: Jawaharlal Nehru Technological University Hyderabad, 2025.