**INTELLIGENT CPU SCHEDULER SIMULATOR**

**A PROJECT REPORT**

# *Submitted by*

# V.PRUDVISWAR

**Reg no: 12316480**

**Section: K23GS**

**Roll No: K23GS39**

# PULPIPATI SAI AYYAPPA

**Reg no: 12315867**

**Section: K23GS**

**Roll No: K23GS56**

# KYATHAM PHANINDRA REDDY

**Reg no: 12320622**

**Section: K23GS**

## Roll No: K23GS34

***In partial fulfillment of the requirements for the degree***

***of***

**BACHELOR OF TECHNOLOGY IN**

# COMPUTER SCIENCE AND ENGINEERING



# Lovely Professional University, Punjab

**APRIL, 2025**

## DECLARATION

I, Vatam Prudvi Swar Reddy, a student pursuing a Bachelor of Technology degree under the CSE discipline at Lovely Professional University, Punjab, hereby declare that all the information furnished in this project report is genuine and based on my work.

Vatam prudvi swar reddy

Registration Number: 12316480

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S. NO.** | **CONTENTS** | **PAGE NO.** |
| 1. | Project-Overview | 4 |
| 2. | Module-Wise Breakdown | 5 |
| 3. | Functionalities | 6 |
| 4. | Technology Used | 7 |
| 5. | Flow Diagram | 8 |
| 6. | Revision Tracking on GitHub | 9 |
| 7. | Conclusion and Future Scope | 9 |
| 8. | References | 10 |
|  | Appendix | 11 |

## I. Project Overview

**1.1 Objective:**

To develop a CLI based Intelligent CPU scheduler simulator, which will implements the scheduling algorithms (FCFS, SJF, Round Robin, Priority Scheduling). Along with real-time visualization and performance metrics for evaluation.

**1.2 Description:**

The CPU Scheduler Simulator is a CLI-based simulator tool , which is designed to simulate and analyze the different CPU scheduling algorithms based on the processes arrival time, burst time, priority and time quantum. And the simulator allows the users to input the process details such as, arrival time, burst time, and priority. With these process details the simulator will visualize the execution time line of processes along with IDLE time lines of CPU through Gantt charts. The simulator will output the process table with computed calculations such as, completion time, waiting time and turnaround time. And It computes the performance metrics such as average waiting time and turnaround time, so it allows users to compare the efficiency of different scheduling techniques.

The simulator provides an interactive simulation of the following CPU scheduling algorithms:

1. First-Come, First-Served (FCFS)
2. Shortest Job First (Non-Preemptive SJF)
3. Shortest Job First (Preemptive SJF)
4. Round Robin (RR)
5. Priority Scheduling (Non-Preemptive)
6. Priority Scheduling (Preemptive)

By implementing these scheduling methods, the simulator enhances the understanding of how operating systems manage the process execution and CPU utilization.

**1.3 Key Features:**

* Supports multiple scheduling algorithms (**FCFS, SJF, RR, Priority Scheduling**).
* Computes **waiting time and turnaround time** for each algorithm.
* Allows **user input** for process details.
* Generates **Gantt charts** to visualize process execution.
* Generates **Process table** for analysis

## II. Module-Wise Breakdown

The Intelligent CPU Scheduler Simulator is structured into multiple modules to ensure the clear separation of the files for better execution. Below is the Breakdown of these modules.

**2.1 Main Module**

The main module acts as the central controller of the complete program to connect with other modules of utility functions along with input and algorithm modules. It provides a menudriven interface for user to interact with scheduler, which allows user to select a scheduling algorithm, input process details and view results. This module calls the appropriate scheduling algorithm and will computes the results from the utility module, and presents the final outputs, including the Gantt chart and performance metrics.

**2.2 Input Handling Module**

The input module is responsible for collecting and validating the user inputs. The module allows the user to input the details such as Process ID, Arrival Time, Burst Time, Priority and Time Quantum (if applicable). The module checks that all the inputs are correctly taken, formatted and within the constraints. Once the validation completed, the process data is stored in a structured format in a C++ structure, and stores these details for further processing by the scheduling algorithms.

**2.3 Scheduling Algorithm Module**

The Scheduling Algorithm module contains all the implementations of all the CPU scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF) Preemptive and Non-Preemptive, Round Robin (RR), and Priority Scheduling both Preemptive and NonPreemptive. This module will execute the processes based on the order of the processes.

**2.4 Utility Module**

This module handles all the visualization of Gantt Charts and Performance Metrics along with process tables.

Gantt Chart representation of process execution.

Process Table, which presents process execution details

Performance Metrics of process execution with average waiting time and turnaround time.

## III. Functionalities

The Intelligent CPU Scheduler Simulator provides various functionalities that which allow the users to input the process details and execute scheduling algorithms with CLI base visual results, and analyses the performance metrics along with process table. These features help in understanding and comparing different CPU scheduling techniques.

**3.1 Key Functionalities:**

1. **User Input Handling:**

The program allows the users to input the process details like Process Id, Arrival time, Burst time, Priority and Time Quantum, It will store these in a structure and give access to the scheduling algorithms to compute the processes based on these inputs.

1. **CPU Algorithm Execution:**

The all CPU Scheduling algorithms such as FCFS, SJF, Round Robin, Priority scheduling are implemented with C++ program, which determines the execution order of processes based on the chosen algorithms and calculates the completion time, waiting time, turnaround time and response time for each process. For Round Robin algorithm it executed based on the process order and time quantum.

1. **Gantt Chart Visualization:**

This visualization generate the CLI based gantt chart to display the execution timeline of the processes along with CPU idle times, and helps the users to understand the scheduling sequence and execution flow of the processes.

1. **Process Table Display:**

A table was represented in CLI which represents the structured table showing the execution details of each processes in order, it displays the main parameters such as completion time, waiting time, and turnaround time along with process id, arrival time, burst time and priority.

**5. Performance Evaluation**

A table was represented with computed key performance metrics of the processes such as Average Waiting Time and Average Turnaround Time.

**6. Menu-Driven Interface:**

It provides an CLI interface with menu for easy navigation and selection of typical scheduling algorithm to perform the simulation for user inputed process details, view results, and restart the simulation.

## IV. Technology Used

**4.1 Programming Languages:**

* **C++** -- Used for implementing the complete program of simulator for scheduling algorithms and handling process execution.

**4.2 Libraries:**

The following C++ standard headers were used in the project:

* *iostream* -- For input/output operations.
* *iomanip* -- For formatting output.
* *climits* -- For handling integer limits in scheduling calculations.
* *queue* -- For managing process execution in queue-based scheduling algorithms like Round Robin.
* *vector*  -- For storing and processing scheduling data efficiently.
* Custom Header File (*scheduler.h*) – Created to connect and manage all modules, to ensuring better code organization and maintainability between algorithms.

**4.3 Tools:**

* VS Code -- Used as the primary IDE for writing and debugging the code.
* GCC (MinGW) -- Used for compiling the C++ code in the Windows environment.
* Build Script (build.bat) -- Created for easy compilation and execution of the project on Windows, automating the build process.
* GitHub -- Used for version control and project management, ensuring smooth collaboration and tracking of code changes.

**V.**

**Flow Diagram**

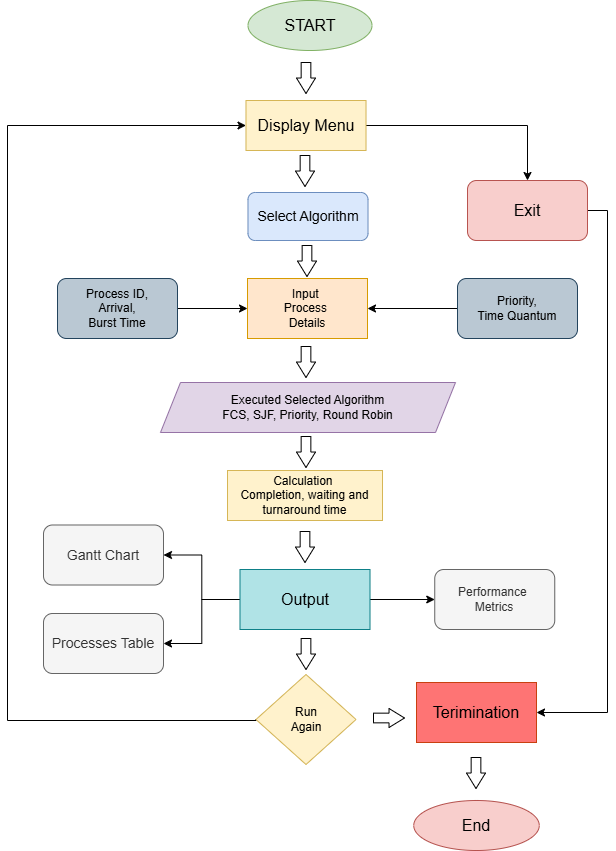


FIG 1. Flow Chart

## VI. Revision Tracking on GitHub

**6.1 Repository Details:**

* **Repository Name: [** cpu\_scheduler\_simulator ]
* **GitHub Link: [** <https://github.com/TanujPalasani/cpu_scheduler_simulator>**]**

**6.2 Revision Tracking and Updates:**

* The project was continuously updated with commits to maintain code improvements and bug fixes.
* Each module was developed and tested separately before merging into the main branch.
* The use of GitHub commit history helped track all changes, making it easier to identify improvements and debug errors.
* The build.bat script and scheduler.h header file were added in later commits to improve compilation efficiency and modularity.
* Added README.md file with complete instructions of the work

## VII. Conclusion and Future Scope

**7.1 Conclusion**

The Intelligent CPU Scheduler Simulator successfully implements multiple CPU scheduling algorithms, including FCFS, SJF, Round Robin, and Priority Scheduling. It provides a menudriven interface that allows users to input process details, execute scheduling algorithms, and analyze performance through Gantt charts, process table and performance metrics. The modular based approach ensures the clarity, accuracy, and ease of use, which making it a valuable tool for users to study and understand better CPU scheduling concepts.

**7.2 Future Scope**

1. **Graphical User Interface (GUI):**

The Intelligent CPU Scheduler Simulator can be further improved by implementing a Graphical User Interface (GUI) to improve user interaction and visualization of scheduling algorithms.

1. **Real-Time Scheduling:**

Expanding the simulator to handle the real-time scheduling algorithms like Rate-Monotonic (RMS) and Earliest Deadline First (EDF).

By integrating with these improvements, the simulator can be a more advanced and interactive tool for studying CPU scheduling techniques.

## VIII. References

**8.1 Textbook Reference:**

* Abraham Silberschatz, Peter B. Galvin, Greg Gagne, *Operating System Concepts, Wiley.*

**8.2 Online Reference:**

* Tutorialspoint – CPU Scheduling [*https://www.tutorialspoint.com/operating\_system/os\_process\_scheduling\_algorithms. htm*](https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm)

**8.3 Development Tools & IDEs:**

* Visual Studio Code – Used for writing and debugging the code.
* Draw.io – Used for designing the flowchart

## Appendix

**A. AI-Generated Project Elaboration/Breakdown Report**

[ This project report section provides a comprehensive, AI-assisted breakdown of the **Intelligent CPU Scheduler Simulator**, structured to ensure clarity, modularity, and effective functionality.

The simulator is composed of the following core modules:

**A1. Algorithm Module:**

This module houses the implementation of all CPU scheduling algorithms—**First Come**

**First Serve (FCFS), Shortest Job First (SJF), Round Robin, and Priority Scheduling**. Each algorithm is developed as an independent function, taking user-defined process inputs and calculating scheduling orders, completion times, and intermediate outputs.

**A2. Input Module:**

This module is responsible for taking all user inputs including **process ID, arrival time, burst time, and priority** (where applicable). It ensures proper data collection and validation before passing the data to the algorithm module for processing.

**A3. Utility Module:**

The utility module handles various output-related tasks:

* Generates and displays the **Gantt Chart**.
* Displays a **formatted process table** including arrival time, burst time, waiting time, turnaround time, and completion time.
* Computes **performance metrics** like average waiting time, turnaround time, and response time for better algorithm analysis.

**A4. Main Module:**

This is the **controller module** of the application. It provides a **menu-driven interface** allowing users to:

* Select the desired scheduling algorithm.
* Input process data.
* View the output results interactively.

It integrates all the above modules and manages program flow, interaction, and logic switching.

**A5. Header File – scheduler.h:**

A custom header file created to define **function declarations** and establish **inter-module communication**. It helps organize code and supports reusability by connecting the implementation across all modules in a clean and structured manner.

**A6. Additional Tools & Structure:**

* **Compiler:** GCC (MinGW)
* **Editor:** Visual Studio Code
* **Build Automation:** build.bat file for compiling multiple modules in one go
* **Version Control:** GitHub for tracking progress and maintaining the codebase
* **Flow Design:** Flow diagram created using Draw.io for visualization

This modular design, powered by AI-based planning and documentation, ensured that the simulator was both easy to develop and scalable for future enhancements.]

### B. Problem Statement

**[ Intelligent CPU Scheduler Simulator Description:** Develop a simulator for CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority Scheduling) with real-time visualizations. The simulator should allow users to input processes with arrival times, burst times, and priorities and visualize Gantt charts and performance metrics like average waiting time and turnaround time.]

**C. Solution/Code**

**C1. Code of Main Module:**

|  |
| --- |
| *#include <iostream>*  *#include "scheduler.h"*    *using namespace std;*    *void displayMenu() {*  *cout << "\n========== Intelligent CPU Scheduler Simulator ==========\n"; cout << "\n1. First Come First Serve (FCFS)\n"; cout << "2. Shortest Job First (SJF Non-Preemptive)\n"; cout << "3. Shortest Job First (SJF Preemptive)\n"; cout << "4. Priority Scheduling (Non-Preemptive)\n"; cout << "5. Priority Scheduling (Preemptive)\n"; cout << "6. Round Robin\n";*  *cout << "7. Exit\n"; cout << "\n";*  *}*    *string getAlgorithmName(int choice) {*  *switch (choice) {*  *case 1: return "First Come First Serve (FCFS)"; case 2: return "Shortest Job First (Non-Preemptive)"; case 3: return "Shortest Job First (Preemptive)"; case 4: return "Priority Scheduling (Non-Preemptive)"; case 5: return "Priority Scheduling (Preemptive)"; case 6: return "Round Robin";*  *default: return "";*  *}*  *}*    *bool confirmChoice(string message) { char confirm; while (true) {*  *cout << message << " (Y/N): ";*  *cin >> confirm;*  *if (confirm == 'y' || confirm == 'Y') return true; if (confirm == 'n' || confirm == 'N') return false;*  *cout << "Invalid input! Please enter 'Y' for Yes or 'N' for No: ";*  *}*  *}*    *int main () {*  *int n, choice, timeQuantum;*    *do { displayMenu();*    *// Get user choice with validation cout << "\nEnter your choice: ";* |

|  |
| --- |
| *while (!(cin >> choice) || choice < 1 || choice > 7) {*  *cout << "Invalid choice! Please enter a number between 1 and 7: "; cin.clear();*  *cin.ignore(1000, '\n');*  *}*    *// Exit program if (choice == 7) {*  *cout << "\nExiting the program...\n"; break;*  *}*    *// Confirm the algorithm choice*  *if (!confirmChoice("\nYou have chosen: " + getAlgorithmName(choice) + ". Do you want to continue?")) continue;*    *// Enter the number of processes with confirmation do {*  *cout << "\nEnter the number of processes: ";*  *while (!(cin >> n) || n <= 0) {*  *cout << "Invalid input! Please enter a positive number: "; cin.clear();*  *cin.ignore(1000, '\n');*  *}*  *} while (!confirmChoice("You have entered " + to\_string(n) + " processes. Do you want to continue?"));*    *Process processes[n];*  *vector<GanttChartEntry> ganttChart;*    *bool requiresPriority = (choice == 4 || choice == 5); processesInput(processes, n, requiresPriority);*    *if (choice == 6) {*  *cout << "Enter Time Quantum: "; while (!(cin >> timeQuantum) || timeQuantum <= 0) { cout << "Invalid input! Please enter a positive number: "; cin.clear(); cin.ignore(1000, '\n');*  *}*  *}*    *// Call selected scheduling algorithm switch (choice) {*  *case 1: FCFS(processes, n, ganttChart); break; case 2: SJF\_NonPreemptive(processes, n, ganttChart); break; case 3: SJF\_Preemptive(processes, n, ganttChart); break; case 4: Priority\_NonPreemptive(processes, n, ganttChart); break; case 5: Priority\_Preemptive(processes, n, ganttChart); break; case 6: RoundRobin(processes, n, timeQuantum, ganttChart); break;*  *}*    *// Display results*  *ganttChartPrint(ganttChart); printProcessTable(processes, n); performanceMetrics(processes, n);*    *} while (true);* |

*return 0;*

*}*

**C2. Codes of Input Module:**

|  |
| --- |
| #include "scheduler.h"  #include <iostream>    using namespace std;    void processesInput(Process processes[], int n, bool isPriorityScheduling) { cout << "\nEnter process details:\n";    for (int i = 0; i < n; i++) { processes[i].pid = i + 1;  cout << "P" << i + 1 << ":\n";    // Arrival Time Validation cout << " Arrival Time: ";  while (!(cin >> processes[i].arrivalTime) || processes[i].arrivalTime < 0) {  cout << " Invalid input! Please enter a valid non-negative number for Arrival Time: "; cin.clear();  cin.ignore(1000, '\n');  }    // Burst Time Validation cout << " Burst Time: ";  while (!(cin >> processes[i].burstTime) || processes[i].burstTime <= 0) { cout << " Invalid input! Please enter a valid positive number for Burst Time: "; cin.clear();  cin.ignore(1000, '\n');  }    // Priority Validation (only for priority scheduling) if (isPriorityScheduling) { cout << " Priority: ";  while (!(cin >> processes[i].priority) || processes[i].priority < 0) {  cout << " Invalid input! Please enter a valid non-negative number for Priority: "; cin.clear();  cin.ignore(1000, '\n');  }  } else {  processes[i].priority = 0; // Default priority  }  }  } |

**C3. Code for Header Module:**

|  |
| --- |
| #ifndef SCHEDULER\_H  #define SCHEDULER\_H    #include <iostream>  #include <iomanip>  #include <vector>    using namespace std;    struct Process { int pid; int arrivalTime; int burstTime; int remainingTime; int completionTime; int waitingTime; int turnaroundTime; bool completed; int startTime; int priority;  };    struct GanttChartEntry {  int pid; int startTime;  int endTime;  };    //Input for Scheduling algos  void processesInput(Process processes[], int n, bool isPriorityScheduling = false);    //Functions for Scheduling Algorithms  void FCFS(Process processes[], int n, vector<GanttChartEntry> &ganttChart); void SJF\_NonPreemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart); void SJF\_Preemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart); void RoundRobin(Process processes[], int n, int timeQuantum, vector<GanttChartEntry>  &ganttChart); void Priority\_NonPreemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart); void Priority\_Preemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart);    //Utility functions for Scheduling Algorithms  void ganttChartPrint(vector<GanttChartEntry> ganttChart); void printProcessTable(Process processes[], int n); void performanceMetrics(Process processes[], int n);    #endif |

**C4. Codes for Algorithm Module:**

FCFS Algorithm:

|  |
| --- |
| #include "scheduler.h"    void FCFS(Process processes[], int n, vector<GanttChartEntry> &ganttChart) { int currentTime = processes[0].arrivalTime; // Track CPU time    for (int i = 0; i < n; i++) {    // Handle CPU idle time  if (processes[i].arrivalTime > currentTime) {  ganttChart.push\_back({-1, currentTime, processes[i].arrivalTime}); // IDLE block currentTime = processes[i].arrivalTime; // It will move time forward  }    // Process execution  processes[i].completionTime = currentTime + processes[i].burstTime; processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime; processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;    // Add to Gantt Chart  ganttChart.push\_back({processes[i].pid, currentTime, processes[i].completionTime});    // Update current time  currentTime = processes[i].completionTime;  }  } |

**SJF Non-Preemptive**

#include <iostream>

#include "scheduler.h"

#include <limits.h> // For INT\_MAX

using namespace std;

void SJF\_NonPreemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart)

{

int currentTime = processes [0] . arrivalTime, completedProcesses = 0;

// Mark all processes as not completed

|  |
| --- |
| for (int i = 0; i < n; i++) {  processes[i].completed = false;  }    while (completedProcesses < n) {  int minBurst = INT\_MAX, selected = -1;    // Find the shortest job that has arrived for (int i = 0; i < n; i++) {  if (!processes[i].completed && processes[i].arrivalTime <= currentTime && processes[i].burstTime < minBurst) { minBurst = processes[i].burstTime; selected = i;  }  }    if (selected == -1) {  // If no process is ready, CPU is idle int nextArrival = INT\_MAX;    // Find the next arriving process for (int i = 0; i < n; i++) {  if (!processes[i].completed && processes[i].arrivalTime > currentTime) { nextArrival = min(nextArrival, processes[i].arrivalTime);  }  }    // Add idle time to Gantt Chart  ganttChart.push\_back({-1, currentTime, nextArrival}); currentTime = nextArrival;  } else {  // Process execution  ganttChart.push\_back({processes[selected].pid, currentTime, currentTime + processes[selected].burstTime});    processes[selected].completionTime = currentTime + processes[selected].burstTime;  processes[selected].turnaroundTime = processes[selected].completionTime - processes[selected].arrivalTime;  processes[selected].waitingTime = processes[selected].turnaroundTime - processes[selected].burstTime;  processes[selected].completed = true;    // Move time forward & increment completed count currentTime = processes[selected].completionTime; completedProcesses++;  } |

}

}

SJF Preemptive

|  |
| --- |
| #include "scheduler.h"  #include <climits>    void SJF\_Preemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart) { int currentTime = processes[0].arrivalTime, completedProcesses = 0; int lastExecutedPid = -1, timeSliceStart = 0;    // Initialize remaining time for all processes for (int i = 0; i < n; i++) {  processes[i].remainingTime = processes[i].burstTime; processes[i].completed = false;  }    while (completedProcesses < n) {  int shortestIndex = -1;  int minRemainingTime = INT\_MAX;    // Find the process with the shortest remaining time that has arrived for (int i = 0; i < n; i++) {  if (!processes[i].completed && processes[i].arrivalTime <= currentTime && processes[i].remainingTime  < minRemainingTime) {  minRemainingTime = processes[i].remainingTime; shortestIndex = i;  }  }    // If no process is available, CPU stays idle if (shortestIndex == -1) { if (lastExecutedPid != -1) {  ganttChart.push\_back({lastExecutedPid, timeSliceStart, currentTime}); // Close last process  }  if (ganttChart.empty() || ganttChart.back().pid != -1) {  ganttChart.push\_back({-1, currentTime, currentTime + 1}); // Mark idle  } else {  ganttChart.back().endTime++; // Extend idle period  }  lastExecutedPid = -1; // Reset last executed process currentTime++;  continue;  }    // Process switching: Record the last executed process in Gantt Chart if (processes[shortestIndex].pid != lastExecutedPid) { if (lastExecutedPid != -1) { ganttChart.push\_back({lastExecutedPid, timeSliceStart, currentTime}); // Close previous process  }  timeSliceStart = currentTime; lastExecutedPid = processes[shortestIndex].pid;  }    // Execute the process for one unit time processes[shortestIndex].remainingTime--; |
| currentTime++;    // If the process completes  if (processes[shortestIndex].remainingTime == 0) {  ganttChart.push\_back({processes[shortestIndex].pid, timeSliceStart, currentTime}); processes[shortestIndex].completionTime = currentTime;  processes[shortestIndex].turnaroundTime = processes[shortestIndex].completionTime - processes[shortestIndex].arrivalTime;  processes[shortestIndex].waitingTime = processes[shortestIndex].turnaroundTime - processes[shortestIndex].burstTime;  processes[shortestIndex].completed = true; completedProcesses++;  lastExecutedPid = -1; // Reset last executed process  }  }  } |

Priority Non-Preemptive

|  |
| --- |
| #include <climits> // For INT\_MAX  #include "scheduler.h"    void Priority\_NonPreemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart) { int currentTime = processes[0].arrivalTime; int completedProcesses = 0;    // Initialize all processes  for (int i = 0; i < n; i++) {  processes[i].completed = false;  processes[i].remainingTime = processes[i].burstTime;  }    while (completedProcesses < n) { int selectedIndex = -1;  int highestPriority = INT\_MAX;    // Find the highest priority available process for (int i = 0; i < n; i++) {  if (!processes[i].completed && processes[i].arrivalTime <= currentTime) {  // Lower priority number = higher priority if (processes[i].priority < highestPriority) { highestPriority = processes[i].priority; selectedIndex = i;  }  // Tie-breaker: Earlier arrival time  else if (processes[i].priority == highestPriority && processes[i].arrivalTime < processes[selectedIndex].arrivalTime) { selectedIndex = i;  }  }  }    if (selectedIndex == -1) { // No process is available, CPU is idle if (!ganttChart.empty() && ganttChart.back().pid == -1) { ganttChart.back().endTime++; // Extend last idle block |
| } else {  ganttChart.push\_back({-1, currentTime, currentTime + 1}); // Start new idle block  }  currentTime++; // Move time forward  } else {  // If there was an idle period before, ensure it ends properly if (!ganttChart.empty() && ganttChart.back().pid == -1) {  ganttChart.back().endTime = currentTime; // Close idle block  }    // Execute the selected process to completion processes[selectedIndex].startTime = currentTime;  ganttChart.push\_back({processes[selectedIndex].pid, currentTime, currentTime + processes[selectedIndex].burstTime});  currentTime += processes[selectedIndex].burstTime;    // Calculate metrics  processes[selectedIndex].completionTime = currentTime;  processes[selectedIndex].turnaroundTime = processes[selectedIndex].completionTime - processes[selectedIndex].arrivalTime;  processes[selectedIndex].waitingTime = processes[selectedIndex].turnaroundTime - processes[selectedIndex].burstTime;  processes[selectedIndex].completed = true; completedProcesses++;  } } } |

Priority Preemptive

|  |
| --- |
| #include "scheduler.h"  #include <climits>    void SJF\_Preemptive(Process processes[], int n, vector<GanttChartEntry> &ganttChart) { int currentTime = processes[0].arrivalTime, completedProcesses = 0; int lastExecutedPid = -1, timeSliceStart = 0;    // Initialize remaining time for all processes for (int i = 0; i < n; i++) {  processes[i].remainingTime = processes[i].burstTime; processes[i].completed = false;  }    while (completedProcesses < n) {  int shortestIndex = -1;  int minRemainingTime = INT\_MAX;    // Find the process with the shortest remaining time that has arrived for (int i = 0; i < n; i++) {  if (!processes[i].completed && processes[i].arrivalTime <= currentTime && processes[i].remainingTime < minRemainingTime) { |

|  |
| --- |
| minRemainingTime = processes[i].remainingTime; shortestIndex = i;  }  }    // If no process is available, CPU stays idle if (shortestIndex == -1) { if (lastExecutedPid != -1) {  ganttChart.push\_back({lastExecutedPid, timeSliceStart, currentTime}); // Close last process  }  if (ganttChart.empty() || ganttChart.back().pid != -1) {  ganttChart.push\_back({-1, currentTime, currentTime + 1}); // Mark idle  } else {  ganttChart.back().endTime++; // Extend idle period  }  lastExecutedPid = -1; // Reset last executed process currentTime++;  continue;  }    // Process switching: Record the last executed process in Gantt Chart if (processes[shortestIndex].pid!= lastExecutedPid) { if (lastExecutedPid != -1) {  ganttChart.push\_back({lastExecutedPid, timeSliceStart, currentTime}); // Close previous process  }  timeSliceStart = currentTime;  lastExecutedPid = processes[shortestIndex].pid;  }    // Execute the process for one unit time processes[shortestIndex].remainingTime--; currentTime++;    // If the process completes  if (processes[shortestIndex].remainingTime == 0) {  ganttChart.push\_back({processes[shortestIndex].pid, timeSliceStart, currentTime}); processes[shortestIndex].completionTime = currentTime; processes[shortestIndex].turnaroundTime =  processes[shortestIndex].completionTime - processes[shortestIndex].arrivalTime; processes[shortestIndex].waitingTime = processes[shortestIndex].turnaroundTime - processes[shortestIndex].burstTime;  processes[shortestIndex].completed = true; completedProcesses++;  lastExecutedPid = -1; // Reset last executed process  } |

}

}

Round Robin

|  |
| --- |
| #include "scheduler.h"  #include <queue>  #include <vector>  #include <climits>    using namespace std;    void RoundRobin(Process processes[], int n, int quantum, vector<GanttChartEntry>  &ganttChart) {  if (quantum <= 0) {// Input validation  cerr << "Error: Time quantum must be positive\n"; return;  }    queue<int> readyQueue; vector<int> remainingTime(n); vector<bool> isQueued(n, false);  int currentTime = processes[0].arrivalTime, completedProcesses = 0;    // Initialize  for (int i = 0; i < n; i++) {  remainingTime[i] = processes[i].burstTime; if (processes[i].arrivalTime == currentTime) { readyQueue.push(i);  isQueued[i] = true;  }  }    while (completedProcesses < n) { if (readyQueue.empty()) { // Handle idle CPU time int nextArrival = INT\_MAX; for (int i = 0; i < n; i++) { if (remainingTime[i] > 0) {  nextArrival = min (nextArrival, processes[I ].arrivalTime);  }  }  if (nextArrival == INT\_MAX) break; // All done    // Add idle period to Gantt chart if (currentTime < nextArrival) {  ganttChart.push\_back({-1, currentTime, nextArrival});  } |
| currentTime = nextArrival;    // Queue arriving processes  for (int i = 0; i < n; i++) {  if (!isQueued[i] && remainingTime[i] > 0 && processes[i].arrivalTime <= currentTime) { readyQueue.push(i);  isQueued[i] = true;  }  }  continue;  }    // Process execution int idx = readyQueue.front(); readyQueue.pop();  isQueued[idx] = false;    int execTime = min (quantum, remainingTime[idx]);  ganttChart.push\_back({processes[idx].pid, currentTime, currentTime + execTime}); currentTime += execTime; remainingTime[idx] -= execTime;    // Check new arrivals during execution for (int i = 0; i < n; i++) {  if (!isQueued[i] && remainingTime[i] > 0 &&  processes[i].arrivalTime > currentTime - execTime && processes[i].arrivalTime <= currentTime) { readyQueue.push(i);  isQueued[i] = true;  }  }    // Requeue if not finished if (remainingTime[idx] > 0) { readyQueue.push(idx);  isQueued[idx] = true;  } else {  processes[idx].completionTime = currentTime;  processes[idx].turnaroundTime = currentTime - processes[idx].arrivalTime; processes[idx].waitingTime = processes[idx].turnaroundTime - processes[idx].burstTime;  completedProcesses++;  }  }  } |

**C5. Codes for Utility Module:**

Gantt Chart:

|  |
| --- |
| #include "scheduler.h"  #include <iomanip>  #include <iostream>    using namespace std;    void ganttChartPrint(std::vector<GanttChartEntry> ganttChart) { if (ganttChart.empty()) return;    cout << "\nGantt Chart:\n";    // Top border  cout << " ";      for (const auto& entry: ganttChart) {  cout << "-------";  (void)entry;  }  cout << "\n|";    // Process IDs  for (const auto& entry: ganttChart) {  if (entry.pid == -1) {  cout << " IDLE |"; // Mark idle periods  } else {  cout << " P" << entry.pid << " |";  }  }    // Bottom border cout << "\n ";  for (const auto& entry: ganttChart) {  cout << "-------";  (void)entry;  }  // Time labels (handles idle start time) cout << "\n" << ganttChart[0].startTime; for (const auto& entry: ganttChart) { cout << setw(7) << entry.endTime;  }  cout << "\n"; |

}

Process Table:

|  |
| --- |
| #include "scheduler.h"  #include <iomanip>  #include <iostream>    void printProcessTable(Process processes[], int n) { std::cout << "\nProcess Table:\n";  std::cout << "-------------------------------------------------------------------------------------------------\n";  std::cout << "| PID | Arrival Time | Burst Time | Priority | Completion Time | Turnaround Time | Waiting Time |\n";  std::cout << "-------------------------------------------------------------------------------------------------\n";    for (int i = 0; i < n; i++) {  std::cout << "| " << std::setw(3) << processes[i].pid << " | " << std::setw(12) << processes[i].arrivalTime  << " | " << std::setw(10) << processes[i].burstTime  << " | " << std::setw(8) << processes[i].priority  << " | " << std::setw(15) << processes[i].completionTime  << " | " << std::setw(15) << processes[i].turnaroundTime  << " | " << std::setw(12) << processes[i].waitingTime << " |\n";  }  std::cout << "-------------------------------------------------------------------------------------------------\n";  } |

Performance Metrics:

|  |
| --- |
| #include "scheduler.h"  #include <iomanip>  #include <iostream>    using namespace std;    void performanceMetrics(Process processes[], int n) { if (n == 0) { cout << "\nNo processes available.\n"; return;  }    float totalTAT = 0, totalWT = 0;  for (int i = 0; i < n; i++) {  totalTAT += processes[i].turnaroundTime; totalWT += processes[i].waitingTime;  }    cout << "\nPerformance Metrics\n";  cout << "---------------------------------------\n"; cout << "| Metric | Value |\n"; cout << "---------------------------------------\n";  cout << "| Average Turn Around Time | " << setw(5) << fixed << setprecision(2) << totalTAT / n << " ms |\n"; cout << "| Average Waiting Time | " << setw(5) << fixed << setprecision(2) << totalWT / n << " ms |\n"; cout << "---------------------------------------\n";  } |