A logo for a university

Description automatically generated**East West University**

**Mini Project Report**

**Semester:** Fall-2024

**Course Title:** Operating System **Course Code:** CSE325 **Sec:** 07

**Submitted by-**

Name: Sheikh Sarafat Hossain

Id: 2022-3-60-109

**Submitted to-**

Prof. Dr. Md. Motaharul Islam

Professor

Department of Computer Science & Engineering

East West University

**Date of Submission:** 19-01-2025

**Mini Project**

**Title: CPU Scheduling Algorithm Simulator and Evaluator**

**Description:** You are required to develop a CPU Scheduling Algorithm Simulator and Evaluator software using C/C++. The software should simulate and compare different CPU scheduling algorithms, calculate the average waiting time, generate a Gantt chart for visual representation.

**Functionality:**

This software is designed to simulate and compare four CPU scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. It calculates key metrics such as waiting time and turnaround time for each process and presents the results. The program allows users to input the burst time for processes and, in the case of Priority Scheduling, the priority of each process. It then runs the algorithms and displays the results including average waiting time, average turnaround time, and a Gantt chart for each algorithm.

**Design Decisions:**

1. **Process Structure**: A structure (struct Process) is used to store the ID, burst time, waiting time, turnaround time, and priority for each process.
2. **Algorithm Implementations**:
   * **FCFS (First-Come, First-Served)**: This non-preemptive algorithm processes tasks in the order they arrive. It calculates waiting and turnaround times based on the sequence of burst times.
   * **SJF (Shortest Job First)**: This non-preemptive algorithm sorts processes based on burst time and calculates the waiting and turnaround times accordingly.
   * **Round Robin**: This preemptive algorithm executes each process for a fixed time quantum. If a process requires more time than the quantum, it is placed back in the queue. The algorithm tracks the remaining burst time for each process and calculates the waiting and turnaround times accordingly.
   * **Priority Scheduling**: This non-preemptive algorithm sorts processes based on priority. Lower priority values represent higher priorities. The algorithm calculates waiting and turnaround times similarly to FCFS but with priority as a deciding factor.
3. **Input/Output**: The program prompts the user to input the number of processes, burst times, and in the case of Priority Scheduling, the priority of each process. After calculating the scheduling results, it displays the average waiting time, average turnaround time, and a Gantt chart to visually represent the process execution sequence.
4. **Comparison of Algorithms**: The software allows the user to compare the performance of the different algorithms, helping in decision-making for scheduling.

**Performance Considerations:**

* **Time Complexity**: The algorithms mainly involve sorting processes (e.g., SJF and Priority Scheduling), which has a time complexity of O(n log n), and simple iterations over processes, which is O(n).
* **Space Complexity**: The space complexity is O(n), as the process data is stored in an array of structures.

**Output:**

Given that the program handles the scheduling of processes, sample input and corresponding output are as follows:

A screenshot of a computer program

Description automatically generated

A computer screen with white text

Description automatically generated

**Full Code:**

#include <stdio.h>

#include <stdlib.h>

struct Process {

    int id;

    int burst\_time;

    int waiting\_time;

    int turnaround\_time;

    int priority; // For priority scheduling

};

// Function prototypes

void fcfs(struct Process processes[], int n);

void sjf(struct Process processes[], int n);

void round\_robin(struct Process processes[], int n, int time\_quantum);

void priority\_scheduling(struct Process processes[], int n);

void calculate\_waiting\_time(struct Process processes[], int n);

void calculate\_turnaround\_time(struct Process processes[], int n);

void display\_gantt\_chart(struct Process processes[], int n);

void compare\_algorithms(struct Process processes[], int n);

int main() {

    int choice, n, time\_quantum;

    printf("Enter number of processes: ");

    scanf("%d", &n);

    struct Process processes[n];

    // Input process burst times and priorities

    for (int i = 0; i < n; i++) {

        printf("Enter burst time for process %d: ", i + 1);

        scanf("%d", &processes[i].burst\_time);

        processes[i].id = i + 1;

        processes[i].waiting\_time = 0;

        processes[i].turnaround\_time = 0;

        processes[i].priority = 0;  // Default priority, will be set for priority scheduling

    }

    while (1) {

        printf("\n1. FCFS\n2. SJF\n3. Round Robin\n4. Priority Scheduling\n5. Compare Algorithms\n6. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1: fcfs(processes, n); break;

            case 2: sjf(processes, n); break;

            case 3:

                printf("Enter time quantum: ");

                scanf("%d", &time\_quantum);

                round\_robin(processes, n, time\_quantum); break;

            case 4: priority\_scheduling(processes, n); break;

            case 5: compare\_algorithms(processes, n); break;

            case 6: exit(0); break;

            default: printf("Invalid choice\n"); break;

        }

    }

    return 0;

}

// FCFS Algorithm

void fcfs(struct Process processes[], int n) {

    int total\_waiting\_time = 0, total\_turnaround\_time = 0;

    int start\_time = 0;

    // Calculate waiting times

    for (int i = 0; i < n; i++) {

        processes[i].waiting\_time = start\_time;

        start\_time += processes[i].burst\_time;

        total\_waiting\_time += processes[i].waiting\_time;

    }

    // Calculate turnaround times

    calculate\_turnaround\_time(processes, n);

    // Display results

    printf("\nFCFS Scheduling:\n");

    display\_gantt\_chart(processes, n);

    printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / n);

    printf("Average Turnaround Time: %.2f\n", (float)total\_turnaround\_time / n);

}

// SJF Algorithm (Non-preemptive)

void sjf(struct Process processes[], int n) {

    int total\_waiting\_time = 0, total\_turnaround\_time = 0;

    struct Process temp;

    // Sort processes by burst time

    for (int i = 0; i < n-1; i++) {

        for (int j = i+1; j < n; j++) {

            if (processes[i].burst\_time > processes[j].burst\_time) {

                temp = processes[i];

                processes[i] = processes[j];

                processes[j] = temp;

            }

        }

    }

    // Calculate waiting times

    int start\_time = 0;

    for (int i = 0; i < n; i++) {

        processes[i].waiting\_time = start\_time;

        start\_time += processes[i].burst\_time;

        total\_waiting\_time += processes[i].waiting\_time;

    }

    // Calculate turnaround times

    calculate\_turnaround\_time(processes, n);

    // Display results

    printf("\nSJF Scheduling (Non-preemptive):\n");

    display\_gantt\_chart(processes, n);

    printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / n);

    printf("Average Turnaround Time: %.2f\n", (float)total\_turnaround\_time / n);

}

// Round Robin Algorithm

void round\_robin(struct Process processes[], int n, int time\_quantum) {

    int total\_waiting\_time = 0, total\_turnaround\_time = 0;

    int remaining\_burst\_time[n];

    // Initialize remaining burst times

    for (int i = 0; i < n; i++) {

        remaining\_burst\_time[i] = processes[i].burst\_time;

    }

    int time = 0;

    int done = 0;

    while (done < n) {

        for (int i = 0; i < n; i++) {

            if (remaining\_burst\_time[i] > 0) {

                if (remaining\_burst\_time[i] > time\_quantum) {

                    time += time\_quantum;

                    remaining\_burst\_time[i] -= time\_quantum;

                } else {

                    time += remaining\_burst\_time[i];

                    processes[i].waiting\_time = time - processes[i].burst\_time;

                    remaining\_burst\_time[i] = 0;

                    done++;

                }

            }

        }

    }

    // Calculate turnaround times

    calculate\_turnaround\_time(processes, n);

    // Display results

    printf("\nRound Robin Scheduling (Time Quantum = %d):\n", time\_quantum);

    display\_gantt\_chart(processes, n);

    printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / n);

    printf("Average Turnaround Time: %.2f\n", (float)total\_turnaround\_time / n);

}

// Priority Scheduling Algorithm (Non-preemptive)

void priority\_scheduling(struct Process processes[], int n) {

    int total\_waiting\_time = 0, total\_turnaround\_time = 0;

    struct Process temp;

    // Input priorities

    for (int i = 0; i < n; i++) {

        printf("Enter priority for process %d: ", i + 1);

        scanf("%d", &processes[i].priority);

    }

    // Sort processes by priority

    for (int i = 0; i < n-1; i++) {

        for (int j = i+1; j < n; j++) {

            if (processes[i].priority > processes[j].priority) {

                temp = processes[i];

                processes[i] = processes[j];

                processes[j] = temp;

            }

        }

    }

    // Calculate waiting times

    int start\_time = 0;

    for (int i = 0; i < n; i++) {

        processes[i].waiting\_time = start\_time;

        start\_time += processes[i].burst\_time;

        total\_waiting\_time += processes[i].waiting\_time;

    }

    // Calculate turnaround times

    calculate\_turnaround\_time(processes, n);

    // Display results

    printf("\nPriority Scheduling:\n");

    display\_gantt\_chart(processes, n);

    printf("Average Waiting Time: %.2f\n", (float)total\_waiting\_time / n);

    printf("Average Turnaround Time: %.2f\n", (float)total\_turnaround\_time / n);

}

// Function to calculate waiting time for all processes

void calculate\_waiting\_time(struct Process processes[], int n) {

    int total\_waiting\_time = 0;

    for (int i = 0; i < n; i++) {

        total\_waiting\_time += processes[i].waiting\_time;

    }

}

// Function to calculate turnaround time for all processes

void calculate\_turnaround\_time(struct Process processes[], int n) {

    for (int i = 0; i < n; i++) {

        processes[i].turnaround\_time = processes[i].waiting\_time + processes[i].burst\_time;

    }

}

// Function to display the Gantt Chart

void display\_gantt\_chart(struct Process processes[], int n) {

    printf("Gantt Chart: ");

    for (int i = 0; i < n; i++) {

        printf("| P%d ", processes[i].id);

    }

    printf("|\n");

}

// Function to compare algorithms

void compare\_algorithms(struct Process processes[], int n) {

    printf("\nComparing Algorithms...\n");

    // Run each algorithm and compare results

    fcfs(processes, n);

    sjf(processes, n);

    int time\_quantum;

    printf("Enter time quantum for Round Robin: ");

    scanf("%d", &time\_quantum);

    round\_robin(processes, n, time\_quantum);

    priority\_scheduling(processes, n);

}

**Conclusion:**

This program successfully implements and compares multiple CPU scheduling algorithms, providing the user with the ability to simulate and analyze different scheduling strategies. The output includes a Gantt chart and average waiting and turnaround times for each algorithm, helping to make informed decisions on process scheduling.