EAST WEST UNIVERSITY

Mini Project

Course code: CSE325

Title: CPU Scheduling Algorithm Simulator and Evaluator

Submitted To

Prof. Dr. Md. Motaharul Islam

Adjunct Faculty,
Department of Computer Science and Engineering

Submitted by

Md. Ariful Islam Opi

2023-1-60-141

Section-4

Title: CPU Scheduling Algorithm Simulator
Implementation Language: C Programming Language

1. Executive Summary

This project implements a comprehensive CPU scheduling algorithm simulator that demonstrates and analyzes five different scheduling techniques: First-Come-First-Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. The simulator provides detailed performance metrics, including waiting time, turnaround time, response time, and completion time, along with statistical analysis showing minimum, maximum, and average values for each metric.

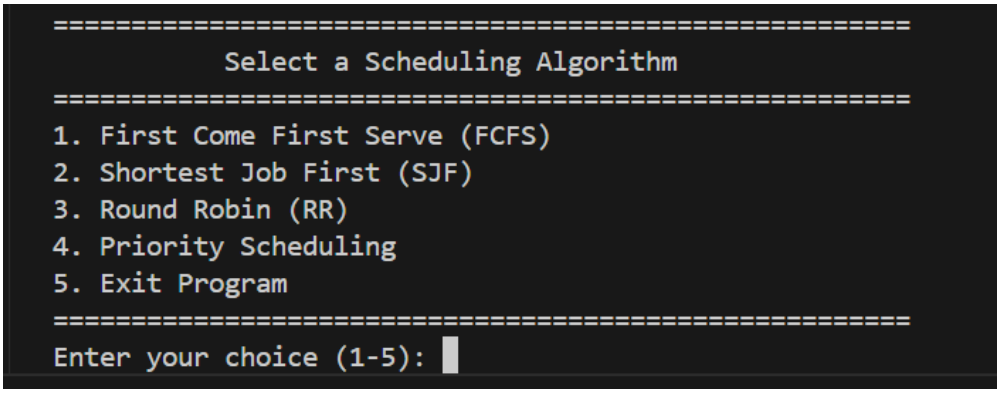
Key Features:

- Implementation of 4 major CPU scheduling algorithms
- User-friendly menu-driven interface
- Comprehensive result visualization with Gantt charts

2.1. System Architecture

```c
// Structure to represent a process
typedef struct {
    int processId;
    int arrivalTime;
    int burstTime;
    int priority;
    int waitingTime;
    int completionTime;
    int turnaroundTime;
} ProcessInfo;
```

2.2. User Interface



3. Algorithm Implementations

3.1 First-Come-First-Serve (FCFS)

Classification: Non-preemptive
Complexity: O(n)

void firstComeFirstServe(ProcessInfo processArray[], int numberOfProcesses)

Key Features:

- Processes execute in strict arrival order
- Simple implementation with no sorting required
- Response time equals waiting time for first process
- No starvation issues but may cause convoy effect

Implementation Details:

- Calculates waiting time cumulatively
- Response time set when process first gets CPU
- Completion time incremented by burst time

Example Output:

```
--- First Come First Serve (FCFS) Scheduling ---

FCFS Scheduling Results:
================================================================================
Process Arrival Burst   Priority        Waiting Turnaround      Completion
================================================================================
P1      0       5       0               0       5               5
P2      1       3       5               4       7               8
P3      2       6       8               6       12              14
P4      2       7       3               12      19              21
================================================================================
Average Waiting Time: 5.50 units
Average Turnaround Time: 10.75 units

Gantt Chart:
============
| P1 | P2 | P3 | P4 |
0   5   8   14   21
```

3.2 Shortest Job First (SJF)

Classification: Non-preemptive
Complexity: O(n^2) due to sorting

void shortestJobFirst(ProcessInfo processArray[], int numberOfProcesses)

Key Features:

- Optimal for minimizing average waiting time
- Uses bubble sort for process ordering
- May cause starvation for longer processes
- Requires knowledge of burst times (assumption)

Implementation Details:

- Sorts processes by burst time using sortProcessesByBurstTime()
- Identical timing calculations to FCFS after sorting
- Maintains process identity through sorting operations

Example Output:

```
--- Shortest Job First (SJF) Scheduling ---

SJF Scheduling Results:
================================================================================
Process Arrival Burst   Priority        Waiting Turnaround      Completion
================================================================================
P2      1       3       5               0       3               4
P1      0       5       0               4       9               9
P3      2       6       8               7       13              15
P4      2       7       3               13      20              22
================================================================================
Average Waiting Time: 6.00 units
Average Turnaround Time: 11.25 units

Gantt Chart:
============
| P2 | P1 | P3 | P4 |
0   3   8   14   21
```

3.3 Round Robin (RR)

Classification: Preemptive
Complexity: O(n) per round

void roundRobinScheduling(ProcessInfo processArray[], int numberOfProcesses, int timeQuantum)

Key Features:

- Fair time sharing among all processes
- User-configurable time quantum
- Prevents starvation through cyclic execution
- Performance highly dependent on quantum size

Implementation Details:

- **Quantum-based execution:** Processes run for full quantum or remaining time
- **Cyclic scheduling:** Continuously cycles through process array
- **Remaining time tracking:** Uses separate array for remaining burst times
- **Fair allocation:** Each process gets equal CPU time opportunity

Example Output:

```
Enter your choice (1-5): 3
Enter time quantum: 2

--- Round Robin Scheduling (Quantum = 2) ---

Round Robin Scheduling Results:
================================================================================
Process Arrival Burst   Priority       Waiting Turnaround      Completion
================================================================================
P1      0       5       0              11      16              16
P2      1       3       5              7       10              11
P3      2       6       8              10      16              18
P4      2       7       3              12      19              21
================================================================================
Average Waiting Time: 10.00 units
Average Turnaround Time: 15.25 units

Gantt Chart:
============
| P1 | P2 | P3 | P4 |
0    5    8    14   21
```

3.4 Priority Scheduling

Classification: Non-preemptive
Complexity: O(n^2) due to sorting

void priorityScheduling(ProcessInfo processArray[], int numberOfProcesses)

Key Features:

- Executes higher priority processes first
- Lower numerical value indicates higher priority
- May cause starvation for low-priority processes
- Suitable for real-time and system processes

Implementation Details:

- Sorts processes using sortProcessesByPriority()
- Priority comparison: processes[i].priority > processes[j].priority
- Identical execution logic to SJF after sorting

Example Output:

```
--- Priority Scheduling ---

Priority Scheduling Results:
================================================================================
Process Arrival Burst   Priority        Waiting Turnaround      Completion
================================================================================
P1      0       5       0               0       5               5
P4      2       7       3               3       10              12
P2      1       3       5               11      14              15
P3      2       6       8               13      19              21
================================================================================
Average Waiting Time: 6.75 units
Average Turnaround Time: 12.00 units

Gantt Chart:
============
| P1 | P4 | P2 | P3 |
0   5   12   15   21
```

4. Timing Calculations

Waiting Time: waitingTime = completionTime - arrivalTime - burstTime
Turnaround Time: turnaroundTime = completionTime - arrivalTime

5. Performance Analysis

5.1 Algorithm Complexity Analysis

Algorithm	Time Complexity	Space Complexity	Preemptive
FCFS	O(n)	O(1)	No
SJF	$O(n^2)$	O(1)	No
RR	O(n×T/Q)	O(n)	Yes
Priority	$O(n^2)$	O(1)	No

Where n = number of processes, T = total execution time, Q = time quantum

5.2 Practical Performance Characteristics

FCFS: Simple but may cause convoy effect
SJF: Optimal average waiting time but potential starvation
RR: Fair allocation with quantum-dependent performance
Priority: Good for mixed workloads with starvation risk

6. Conclusion

The CPU Scheduling Algorithm Simulator successfully demonstrates the fundamental concepts of process scheduling in operating systems. The implementation provides a comprehensive comparison platform for five major scheduling algorithms, enhanced with detailed performance metrics and statistical analysis. The project serves as an effective educational tool for understanding CPU scheduling algorithms and their performance characteristics, providing students with hands-on experience in operating systems concepts and implementation techniques.

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). John Wiley & Sons.
2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. Stallings, W. (2017). *Operating Systems: Internals and Design Principles* (8th ed.). Pearson.

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_PROCESSES 100


// Structure to represent a process

typedef struct {

   int processId;

   int arrivalTime;

   int burstTime;

   int priority;

   int waitingTime;

   int completionTime;

   int turnaroundTime;

} ProcessInfo;


// Function prototypes
```

```
void firstComeFirstServe(ProcessInfo processArray[], int numberOfProcesses);

void shortestJobFirst(ProcessInfo processArray[], int numberOfProcesses);

void roundRobinScheduling(ProcessInfo processArray[], int numberOfProcesses, int
timeQuantum);

void priorityScheduling(ProcessInfo processArray[], int numberOfProcesses);

void displayGanttChart(ProcessInfo processArray[], int numberOfProcesses);

void displayProcessTable(ProcessInfo processArray[], int numberOfProcesses, char
algorithmName[]);

float calculateAverageWaitingTime(ProcessInfo processArray[], int numberOfProcesses);

void sortProcessesByBurstTime(ProcessInfo processArray[], int numberOfProcesses);

void sortProcessesByPriority(ProcessInfo processArray[], int numberOfProcesses);

void copyProcessArray(ProcessInfo source[], ProcessInfo destination[], int
numberOfProcesses);


int main() {
   int numberOfProcesses, userChoice, timeQuantum;
   ProcessInfo originalProcesses[MAX_PROCESSES];
   ProcessInfo workingProcesses[MAX_PROCESSES];

   // Display welcome message
   printf("=====================================================\n");
   printf("      CPU Scheduling Algorithm Simulator\n");
   printf("=====================================================\n\n");

   // Get number of processes from user
   printf("Enter the number of processes (max %d): ", MAX_PROCESSES);
   scanf("%d", &numberOfProcesses);
```

```
    if (numberOfProcesses <= 0 || numberOfProcesses > MAX_PROCESSES) {
        printf("Invalid number of processes! Please enter a value between 1 and %d.\n", MAX_PROCESSES);
        return 1;
    }

    // Input process details
    printf("\nEnter process details:\n");
    printf("Format: [Arrival Time] [Burst Time] [Priority]\n");
    printf("-----------------------------------------------\n");

    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        printf("Process P%d: ", processIndex + 1);
        scanf("%d %d %d",
              &originalProcesses[processIndex].arrivalTime,
              &originalProcesses[processIndex].burstTime,
              &originalProcesses[processIndex].priority);

        originalProcesses[processIndex].processId = processIndex + 1;
        originalProcesses[processIndex].waitingTime = 0;
        originalProcesses[processIndex].completionTime = 0;
        originalProcesses[processIndex].turnaroundTime = 0;
    }

    // Main menu loop
```

```
while (true) {
    printf("\n=====================================================\n");
    printf("         Select a Scheduling Algorithm\n");
    printf("=====================================================\n");
    printf("1. First Come First Serve (FCFS)\n");
    printf("2. Shortest Job First (SJF)\n");
    printf("3. Round Robin (RR)\n");
    printf("4. Priority Scheduling\n");
    printf("5. Exit Program\n");
    printf("=====================================================\n");
    printf("Enter your choice (1-5): ");
    scanf("%d", &userChoice);

    // Create a working copy of processes for each algorithm
    copyProcessArray(originalProcesses, workingProcesses, numberOfProcesses);

    switch (userChoice) {
        case 1:
            firstComeFirstServe(workingProcesses, numberOfProcesses);
            break;
        case 2:
            shortestJobFirst(workingProcesses, numberOfProcesses);
            break;
        case 3:
            printf("Enter time quantum: ");
            scanf("%d", &timeQuantum);
```

```
            if (timeQuantum <= 0) {
                printf("Time quantum must be positive!\n");
                break;
            }
            roundRobinScheduling(workingProcesses, numberOfProcesses, timeQuantum);
            break;
        case 4:
            priorityScheduling(workingProcesses, numberOfProcesses);
            break;
        case 5:
            printf("\nThank you for using CPU Scheduling Simulator!\n");
            exit(0);
        default:
            printf("Invalid choice! Please enter a number between 1-5.\n");
      }
   }

   return 0;
}

// Copy process array to preserve original data
void copyProcessArray(ProcessInfo source[], ProcessInfo destination[], int
numberOfProcesses) {
   for (int i = 0; i < numberOfProcesses; i++) {
      destination[i] = source[i];
   }
```

```
}

// First Come First Serve Scheduling Algorithm
void firstComeFirstServe(ProcessInfo processArray[], int numberOfProcesses) {
   printf("\n--- First Come First Serve (FCFS) Scheduling ---\n");

   int currentTime = 0;

   // Calculate waiting time and completion time
   for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
      if (currentTime < processArray[processIndex].arrivalTime) {
         currentTime = processArray[processIndex].arrivalTime;
      }

      processArray[processIndex].waitingTime = currentTime -
processArray[processIndex].arrivalTime;
      currentTime += processArray[processIndex].burstTime;
      processArray[processIndex].completionTime = currentTime;
      processArray[processIndex].turnaroundTime =
         processArray[processIndex].completionTime -
processArray[processIndex].arrivalTime;
   }

   displayProcessTable(processArray, numberOfProcesses, "FCFS");
}
```

```
// Shortest Job First Scheduling Algorithm
void shortestJobFirst(ProcessInfo processArray[], int numberOfProcesses) {
   printf("\n--- Shortest Job First (SJF) Scheduling ---\n");

   // Sort processes by burst time
   sortProcessesByBurstTime(processArray, numberOfProcesses);

   int currentTime = 0;

   // Calculate waiting time and completion time
   for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
      if (currentTime < processArray[processIndex].arrivalTime) {
         currentTime = processArray[processIndex].arrivalTime;
      }

      processArray[processIndex].waitingTime = currentTime -
processArray[processIndex].arrivalTime;
      currentTime += processArray[processIndex].burstTime;
      processArray[processIndex].completionTime = currentTime;
      processArray[processIndex].turnaroundTime =
         processArray[processIndex].completionTime -
processArray[processIndex].arrivalTime;
   }

   displayProcessTable(processArray, numberOfProcesses, "SJF");
}
```

```
// Round Robin Scheduling Algorithm
void roundRobinScheduling(ProcessInfo processArray[], int numberOfProcesses, int timeQuantum) {
  printf("\n--- Round Robin Scheduling (Quantum = %d) ---\n", timeQuantum);

  int remainingBurstTime[numberOfProcesses];
  int currentTime = 0;

  // Initialize remaining burst times
  for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
    remainingBurstTime[processIndex] = processArray[processIndex].burstTime;
    processArray[processIndex].waitingTime = 0;
  }

  // Process execution loop
  bool allProcessesCompleted = false;
  while (!allProcessesCompleted) {
    allProcessesCompleted = true;

    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
      if (remainingBurstTime[processIndex] > 0) {
        allProcessesCompleted = false;

        if (remainingBurstTime[processIndex] > timeQuantum) {
          // Process runs for full quantum
```

```
            currentTime += timeQuantum;
            remainingBurstTime[processIndex] -= timeQuantum;
        } else {
            // Process completes
            currentTime += remainingBurstTime[processIndex];
            processArray[processIndex].completionTime = currentTime;
            processArray[processIndex].waitingTime =
                currentTime - processArray[processIndex].burstTime -
processArray[processIndex].arrivalTime;
            processArray[processIndex].turnaroundTime =
                processArray[processIndex].completionTime -
processArray[processIndex].arrivalTime;
            remainingBurstTime[processIndex] = 0;
        }
      }
    }
  }

  displayProcessTable(processArray, numberOfProcesses, "Round Robin");
}

// Priority Scheduling Algorithm
void priorityScheduling(ProcessInfo processArray[], int numberOfProcesses) {
  printf("\n--- Priority Scheduling ---\n");

  // Sort processes by priority (lower number = higher priority)
```

```
    sortProcessesByPriority(processArray, numberOfProcesses);

    int currentTime = 0;

    // Calculate waiting time and completion time
    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        if (currentTime < processArray[processIndex].arrivalTime) {
            currentTime = processArray[processIndex].arrivalTime;
        }

        processArray[processIndex].waitingTime = currentTime -
processArray[processIndex].arrivalTime;
        currentTime += processArray[processIndex].burstTime;
        processArray[processIndex].completionTime = currentTime;
        processArray[processIndex].turnaroundTime =
            processArray[processIndex].completionTime -
processArray[processIndex].arrivalTime;
    }

    displayProcessTable(processArray, numberOfProcesses, "Priority");
}

// Sort processes by burst time (ascending order)
void sortProcessesByBurstTime(ProcessInfo processArray[], int numberOfProcesses) {
    for (int i = 0; i < numberOfProcesses - 1; i++) {
        for (int j = i + 1; j < numberOfProcesses; j++) {
```

```
            if (processArray[i].burstTime > processArray[j].burstTime) {
                ProcessInfo temp = processArray[i];
                processArray[i] = processArray[j];
                processArray[j] = temp;
            }
        }
    }
}

// Sort processes by priority (ascending order - lower number = higher priority)
void sortProcessesByPriority(ProcessInfo processArray[], int numberOfProcesses) {
    for (int i = 0; i < numberOfProcesses - 1; i++) {
        for (int j = i + 1; j < numberOfProcesses; j++) {
            if (processArray[i].priority > processArray[j].priority) {
                ProcessInfo temp = processArray[i];
                processArray[i] = processArray[j];
                processArray[j] = temp;
            }
        }
    }
}

// Display process scheduling results in a table format
void displayProcessTable(ProcessInfo processArray[], int numberOfProcesses, char
algorithmName[]) {
    printf("\n%s Scheduling Results:\n", algorithmName);
```

```
    printf("================================================================
========\n");
    printf("Process\tArrival\tBurst\tPriority\tWaiting\tTurnaround\tCompletion\n");
    printf("================================================================
========\n");

    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        printf("P%d\t%d\t%d\t%d\t\t%d\t%d\t\t%d\n",
               processArray[processIndex].processId,
               processArray[processIndex].arrivalTime,
               processArray[processIndex].burstTime,
               processArray[processIndex].priority,
               processArray[processIndex].waitingTime,
               processArray[processIndex].turnaroundTime,
               processArray[processIndex].completionTime);
    }

    printf("================================================================
========\n");

    float averageWaitingTime = calculateAverageWaitingTime(processArray,
numberOfProcesses);
    float averageTurnaroundTime = 0.0;

    for (int i = 0; i < numberOfProcesses; i++) {
        averageTurnaroundTime += processArray[i].turnaroundTime;
    }
```

```
    averageTurnaroundTime /= numberOfProcesses;

    printf("Average Waiting Time: %.2f units\n", averageWaitingTime);
    printf("Average Turnaround Time: %.2f units\n", averageTurnaroundTime);

    displayGanttChart(processArray, numberOfProcesses);
}

// Display Gantt Chart representation
void displayGanttChart(ProcessInfo processArray[], int numberOfProcesses) {
    printf("\nGantt Chart:\n");
    printf("============\n");

    // Print process boxes
    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        printf("| P%d ", processArray[processIndex].processId);
    }
    printf("|\n");

    // Print timeline
    int currentTime = 0;
    printf("0");
    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        currentTime += processArray[processIndex].burstTime;
        printf("   %d", currentTime);
    }
```

```
    printf("\n");
}

// Calculate average waiting time
float calculateAverageWaitingTime(ProcessInfo processArray[], int numberOfProcesses) {
    float totalWaitingTime = 0.0;

    for (int processIndex = 0; processIndex < numberOfProcesses; processIndex++) {
        totalWaitingTime += processArray[processIndex].waitingTime;
    }

    return totalWaitingTime / numberOfProcesses;
}
```