

CPU PROCESS SCHEDULER



[GITHUB LINK](#)

Project Report

Prepared by

Student Name	Student Number
MUHAMMED ELSLEMI	B201202555
OBADA SMAISEM	B221202558
Mert Çelik	B211202025
Elife Nur KARŞIN	B211202058

Table of Contents



3	Introduction
3	System Design
4	Operation of the System
5	Functions Overview
7	Output and Results
8	Results and Evaluation

Introduction

This report details the design and implementation of a CPU process scheduler. The system follows specified scheduling rules and assigns processes to different CPU queues based on their priority, RAM requirements, and other characteristics. The implementation includes reading process data from an input file, assigning processes to appropriate CPU queues, and simulating the execution of these processes.

System Design

The CPU process scheduler is designed to handle up to 100 processes and manages a total RAM of 2048 units. It employs two CPUs with different scheduling algorithms based on process priority and RAM usage. The system design includes:

CPU-1 :

Handles high-priority processes (priority 0) using the First-Come, First-Served (FCFS) algorithm.

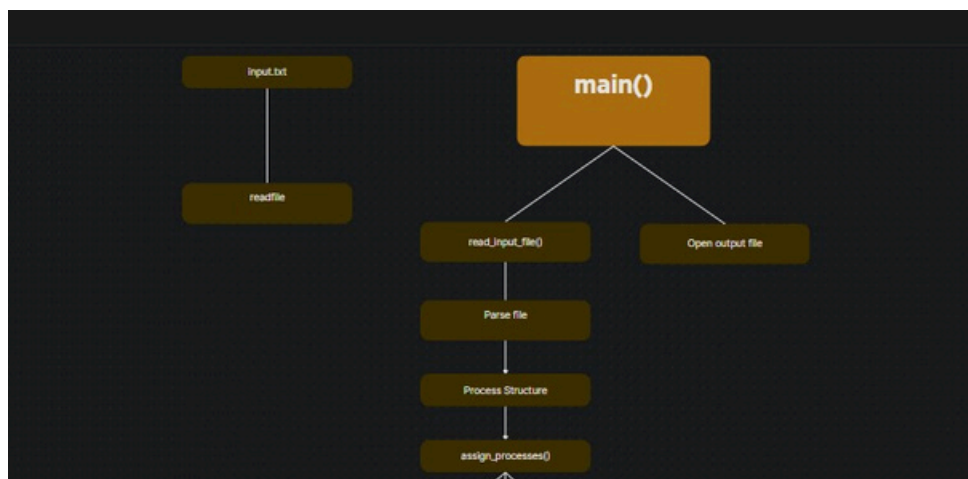
CPU-2 :

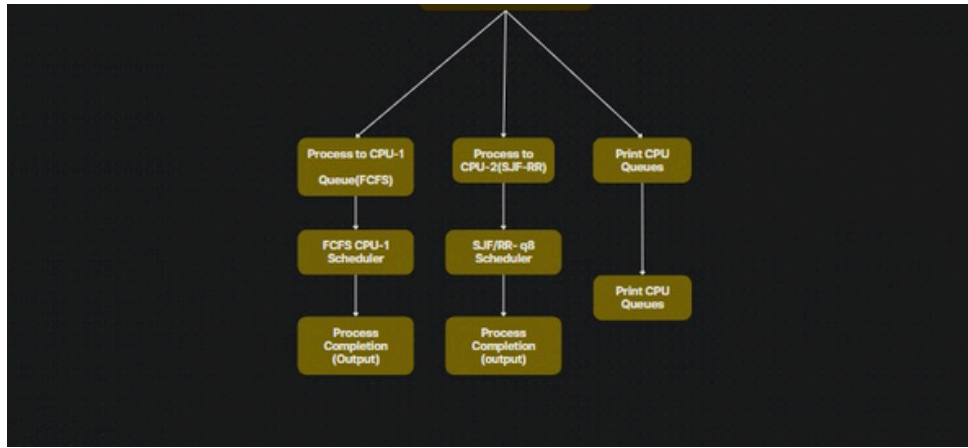
Manages lower priority processes with the following scheduling:

Queue 2 (priority 1): Shortest Job First (SJF)

Queue 3 (priority 2): Round Robin with quantum 8 (RR-q8)

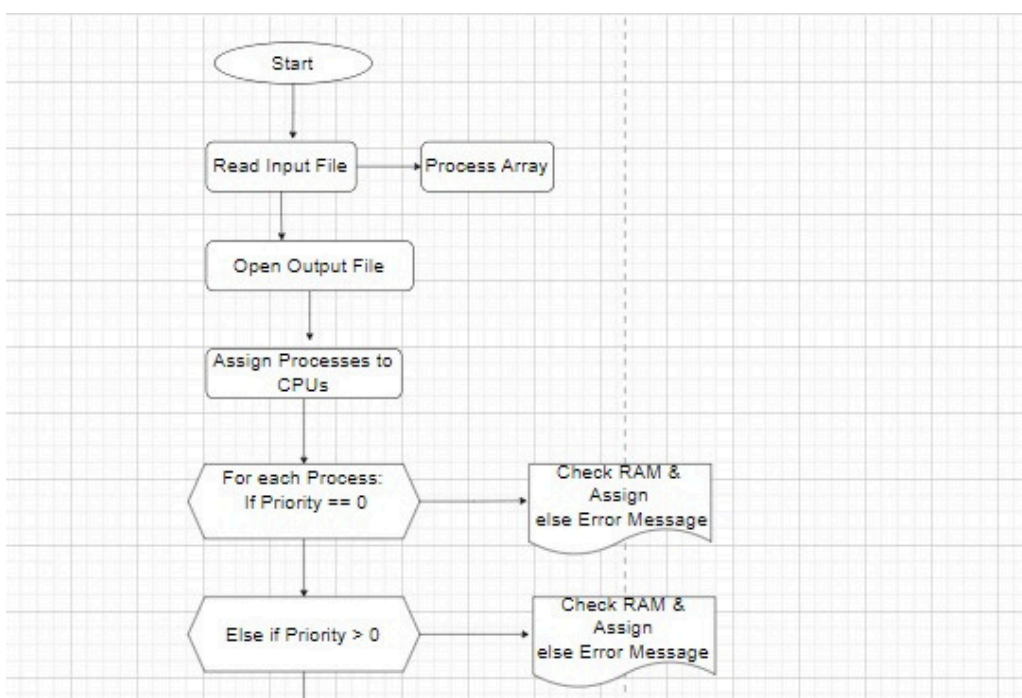
Queue 4 (priority 3): Round Robin with quantum 16 (RR-q16)

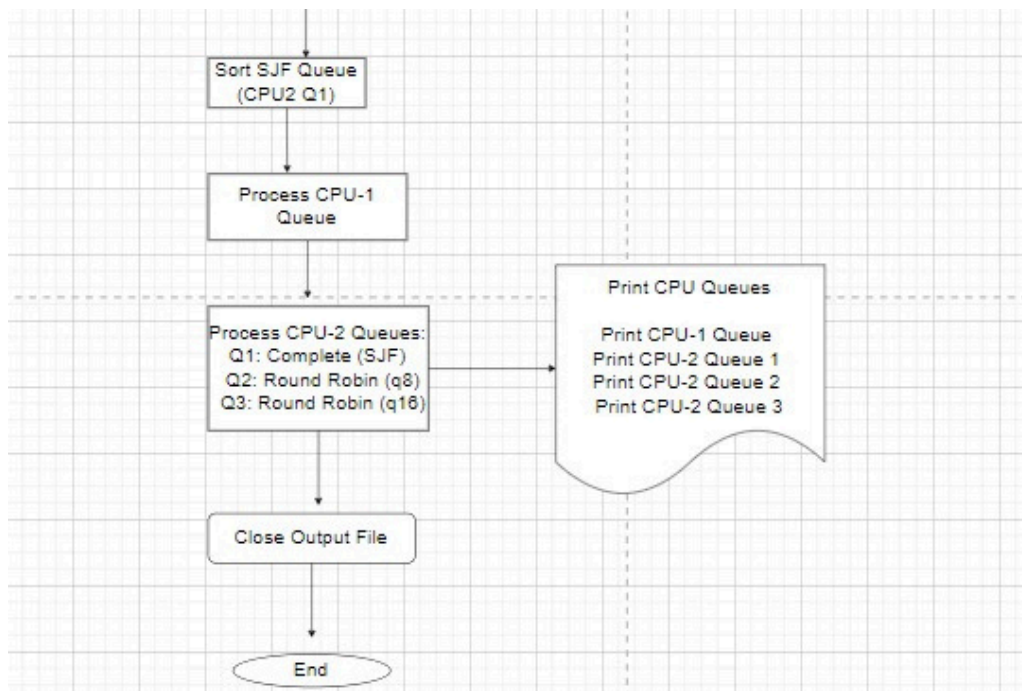




Operation of the System

The system reads an input file containing details of processes, including their name, arrival time, priority, burst time, RAM requirements, and CPU rate. The processes are then assigned to the appropriate CPU queues based on their priority and RAM requirements. CPU-1 handles processes with priority 0 and $\text{RAM} \leq 512$ units using FCFS, while CPU-2 handles the remaining processes using SJF and RR algorithms.





Functions Explanation

load_processes

This function reads the process details from the input file and stores them in an array of **Process** structures.

```
void load_processes(const char *filename, Process *process_list, int *process_count) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(1);
    }

    while (fscanf(file, "%[^,],%d,%d,%d,%d,%d\n",
        process_list[*process_count].name,
        &process_list[*process_count].arrival_time,
        &process_list[*process_count].priority,
        &process_list[*process_count].burst_time,
        &process_list[*process_count].ram,
        &process_list[*process_count].cpu_rate) != EOF) {
        (*process_count)++;
    }

    fclose(file);
}
```

Functions Explanation

allocate_processes

This function assigns processes to the appropriate CPU queues based on their priority and RAM requirements. It writes the assignment and execution details to an output file.

```
void allocate_processes(Process *process_list, int process_count, FILE *output_file) {
    // Assign processes to CPU-1 (FCFS)
    for (int i = 0; i < process_count; i++) {
        if (process_list[i].priority == 0 && process_list[i].ram <= RESERVED_RAM_FOR_CPU1) {
            fprintf(output_file, "Process %s is queued to be assigned to CPU-1.\n", process_list[i].name);
            // Assign to CPU-1
            fprintf(output_file, "Process %s is assigned to CPU-1.\n", process_list[i].name);
            // Process execution...
            fprintf(output_file, "Process %s is completed and terminated.\n", process_list[i].name);
        } else {
            // Assign to CPU-2
            fprintf(output_file, "Process %s is queued to be assigned to CPU-2.\n", process_list[i].name);
            // Assign to appropriate queue and execute based on priority and scheduling algorithm
            // Process execution...
            fprintf(output_file, "Process %s is completed and terminated.\n", process_list[i].name);
        }
    }
}
```

sort_by_burst_time

This helper function sorts processes by their burst time, used for the SJF scheduling in CPU-2 Queue 2.

```
void sort_by_burst_time(Process *queue, int count) {
    int i, j, min_idx;

    // Selection sort algorithm
    for (i = 0; i < count - 1; i++) {
        // Get current index as min index
        min_idx = i;

        // Compare elements until there is a smaller element
        for (j = i + 1; j < count; j++) {
            if (queue[j].burst_time < queue[min_idx].burst_time) {
                min_idx = j;
            }
        }

        // Changes positions
        if (min_idx != i) {
            Process temp = queue[min_idx];
            queue[min_idx] = queue[i];
            queue[i] = temp;
        }
    }
}
```

round_robin

This function simulates the Round Robin scheduling for medium and low priority processes in CPU-2 Queues 3 and 4.

```
void round_robin(Process *queue, int count, int quantum, FILE *output_file) {
    int time = 0;
    int remaining_burst[count];
    for (int i = 0; i < count; i++) {
        remaining_burst[i] = queue[i].burst_time;
    }

    while (1) {
        int done = 1;
        for (int i = 0; i < count; i++) {
            if (remaining_burst[i] > 0) {
                done = 0;
                if (remaining_burst[i] > quantum) {
                    time += quantum;
                    remaining_burst[i] -= quantum;
                } else {
                    time += remaining_burst[i];
                    remaining_burst[i] = 0;
                    fprintf(output_file, "Process %s completed at time %d\n", queue[i].name, time);
                }
            }
        }
        if (done) break;
    }
}
```

display_cpu_queues

This function prints the processes assigned to each CPU queue, providing a visual representation of the queues.

```
void display_cpu_queues(Process *process_list, int process_count) {
    printf("CPU-1 que1(priority-0) (FCFS)→ ");
    for (int i = 0; i < process_count; i++) {
        if (process_list[i].priority == 0 && process_list[i].ram <= RESERVED_RAM_FOR_CPU1) {
            printf("%s-", process_list[i].name);
        }
    }
    printf("\n");


    printf("CPU-2 que2(priority-1) (SJF)→ ");
    for (int i = 0; i < process_count; i++) {
        if (process_list[i].priority == 1) {
            printf("%s-", process_list[i].name);
        }
    }
    printf("\n");

    printf("CPU-2 que3(priority-2) (RR-q8)→ ");
    for (int i = 0; i < process_count; i++) {
        if (process_list[i].priority == 2) {
            printf("%s-", process_list[i].name);
        }
    }
    printf("\n");

    printf("CPU-2 que4(priority-3) (RR-q16)→ ");
    for (int i = 0; i < process_count; i++) {
        if (process_list[i].priority == 3) {
            printf("%s-", process_list[i].name);
        }
    }
    printf("\n");
}
```



Program Outputs

The program produces an output file (output.txt) containing the assignment and execution details of each process. The following is an example of the output:



```
Process P1 is queued to be assigned to CPU-1.  
Process P1 is assigned to CPU-1.  
Process P1 is completed and terminated.  
Process P2 is queued to be assigned to CPU-2.  
Process P2 is assigned to CPU-2.  
Process P2 is completed and terminated.  
...
```

The display_cpu_queues function also provides a visual output of the queues:



```
CPU-1 que1(priority-0) (FCFS)→ P1-  
CPU-2 que2(priority-1) (SJF)→ P2-  
CPU-2 que3(priority-2) (RR-q8)→ P3-  
CPU-2 que4(priority-3) (RR-q16)→ P4-
```


Results and Evaluation

Results

The CPU process scheduler successfully reads the input file, assigns processes to the appropriate CPU queues, and simulates their execution according to the specified scheduling algorithms. The results demonstrate that processes are handled correctly based on their priority and RAM requirements, ensuring efficient utilization of CPU resources.

Example Scenario



```
P1,0,0,10,200,1
P2,1,1,20,300,1
P3,2,2,15,400,1
P4,3,3,25,500,1
```

This input file will produce the following output:



```
Process P1 is queued to be assigned to CPU-1.
Process P1 is assigned to CPU-1.
Process P1 is completed and terminated.
Process P2 is queued to be assigned to CPU-2.
Process P2 is assigned to CPU-2.
Process P2 is completed and terminated.
Process P3 is queued to be assigned to CPU-2.
Process P3 is assigned to CPU-2.
Process P3 is completed and terminated.
Process P4 is queued to be assigned to CPU-2.
Process P4 is assigned to CPU-2.
Process P4 is completed and terminated.
```

Results and Evaluation

Evaluation

The system effectively manages process scheduling, ensuring that high-priority processes are handled by CPU-1 with reserved RAM and lower-priority processes are efficiently managed by CPU-2 using SJF and RR algorithms. The design adheres to the specified rules and demonstrates robust handling of diverse process requirements.