MIT World Peace University

Operating Systems
Second Year B. Tech, Semester 3

Simulation of CPU Scheduling Algorithms - FCFS and Round Robin

Assignment 3
Practical Report

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A2, PA 20

November 3, 2022

1 Code

```c
#include <stdio.h>
#include <math.h>
// Turnaround time - completion time - Arrival time
// Waiting time = Turn around time - Burst time
// Write a program to simulate the first come Non preemptive first come first
    serve cpu scheduling algorithm.

struct Process
{
    int process_id;
    int burst_time;
    int arrival_time;
    int service_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
    int time_quantum;
};

void swap(struct Process *a, struct Process *b)
{
    struct Process temp = *a;
    *a = *b;
    *b = temp;
}

void accept_array(struct Process processes[], int number_of_processes)
{
    for (int i = 0; i < number_of_processes; i++)
    {
        printf("Enter the Process id : ");
        scanf("%d", &processes[i].process_id);
        printf("\nEnter the Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter the Burst Times: ");
        scanf("%d", &processes[i].burst_time);
        printf("\n");
        // assign others to 0
    }
}

void display(struct Process processes[], int number_of_processes, int i)
{

    printf("
    ---------------------------------------------------------------------------------------------\
    n");
    printf("|\tID|\tBur|\tArr|\tser|\twait|\tCom|\tTAT\n");
    for (; i < number_of_processes; i++)
    {
        printf("|\t%d|\t%d|\t%d|\t%d|\t%d|\t%d|\t%d|\n", processes[i].process_id,
    processes[i].burst_time, processes[i].arrival_time, processes[i].service_time,
    processes[i].waiting_time, processes[i].completion_time, processes[i].
    turnaround_time);
    }
    printf("
```

```
    ----------------------------------------------------------------------------------------
    n");
}

void displayGanttChartRR(struct Process processes[], int process_id, int cpu_time)
{
    printf("|%d|\t", cpu_time);
    printf("%d\t", processes[process_id].process_id);
}

void displayGanttChartFSFS(struct Process processes[], int number_of_processes)
{
    for (int i = 0; i < number_of_processes * 4; i++)
    {
        printf("====");
    }
    printf("\n");

    for (int i = 0; i < number_of_processes; i++)
    {
        printf("|\tP%d\t|", processes[i].process_id);
    }
    printf("\n");

    for (int i = 0; i < number_of_processes * 4; i++)
    {
        printf("----");
    }
    printf("\n");

    for (int i = 0; i < number_of_processes; i++)
    {
        printf("|%d|\t\t", processes[i].service_time);
    }
    printf("|%d|\n", processes[number_of_processes - 1].completion_time);
    for (int i = 0; i < number_of_processes * 4; i++)
    {
        printf("====");
    }
    printf("\n");
}

int insertion_sort(struct Process processes[], int number_of_processes)
{
    int j, key;
    for (int i = 0; i < number_of_processes; i++)
    {
        key = processes[i].arrival_time;
        j = i - 1;
        while (j >= 0 && processes[j].arrival_time > key)
        {
            swap(&processes[j + 1], &processes[j]);
            j--;
        }
    }
}

void npfcfs(struct Process processes[], int number_of_processes)
{
```

```
    for (int i = 0; i < number_of_processes; i++)
    {
        processes[i].completion_time = 0;
        processes[i].waiting_time = 0;
        processes[i].service_time = 0;
        processes[i].turnaround_time = 0;
    }
    processes[0].service_time = processes[0].arrival_time;
    processes[0].waiting_time = 0;

    for (int i = 1; i < number_of_processes; i++)
    {
        processes[i].service_time = processes[i - 1].burst_time + processes[i - 1].service_time;
    }
    for (int i = 1; i < number_of_processes; i++)
    {
        for (int j = 0; j < i; j++)
        {
            processes[i].waiting_time += processes[j].burst_time;
        }
        processes[i].waiting_time -= processes[i].arrival_time;
    }
    for (int i = 0; i < number_of_processes; i++)
    {
        processes[i].completion_time = processes[i].service_time + processes[i].burst_time;
    }
    for (int i = 0; i < number_of_processes; i++)
    {
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
    }
}

int find_size_of_ready_queue(struct Process processes[], int number_of_processes, int time_quantum)
{
    int n = 0;
    for (int i = 0; i < number_of_processes; i++)
    {
        n += processes[i].burst_time / time_quantum;
        if (processes[i].burst_time % time_quantum != 0 && processes[i].burst_time > time_quantum)
            n++;
    }
    // printf("%d", n);
    return n;
}

void round_robin(struct Process processes[], int number_of_processes, int time_quantum)
{
    int process_counter = 0;
    // Safely assigning every value we have to calculate to zero.
    for (int i = 0; i < number_of_processes; i++)
    {
        processes[i].completion_time = 0;
        processes[i].waiting_time = 0;
```

```c
        processes[i].service_time = 0;
        processes[i].turnaround_time = 0;
    }
    // This is like a pre defined thing.
    processes[process_counter].service_time = processes[process_counter].
    arrival_time;

    // we need to put things in the ready queue, so we need its size. This
    function calculates that.
    int ready_queue_size = find_size_of_ready_queue(processes, number_of_processes
    , time_quantum);
    int ready_queue_counter = 0;
    int current_node = 0;
    int cpu_time = 0; // to count the cpu clock cycles.

    int ready_queue[ready_queue_size]; // creating the ready queue.

    // Assigning values of that ready queue to -1.
    for (int i = 0; i < ready_queue_size; i++)
    {
        ready_queue[i] = -1;
    }

    // Let us push the first process to the ready queue. This is the one that has
    arrived first as that list is sorted.
    ready_queue[current_node] = process_counter;

    // Creating a copy of the burst times array, as those values have to change in
     preemptive algorithms to keep track of them.
    int burst_times[number_of_processes];
    for (int i = 0; i < number_of_processes; i++)
    {
        burst_times[i] = processes[i].burst_time;
    }

    // Drawing the first line of the Gantt Chart

    for (int i = 0; i < ready_queue_size; i++)
        printf("-------------------");
    printf("\n");

    // Drawing the last line of the Gantt Chart
    // well increment the counter as per as our need, and stop the scheduling when
     we are done with the queue.
    while (current_node <= ready_queue_size)
    {
        displayGanttChartRR(processes, ready_queue[current_node], cpu_time);

        int cpu_cycle_begin_state = cpu_time;

        /// Execution of cpu begins

        // increment the cpu time by time quantum only if the burst time of the
    current process is greater than the time quantum.
        if (burst_times[ready_queue[current_node]] >= time_quantum)
        {
            cpu_time += time_quantum;
            burst_times[ready_queue[current_node]] -= time_quantum;
        }
```

```
        else
        {
            cpu_time += burst_times[ready_queue[current_node]];
            burst_times[ready_queue[current_node]] = 0;
        }

        /// Execution complete

        int cpu_cycle_end_state = cpu_time;

        // check if any processes came in the meanwhile when we incremented the cpu_counter
        for (int i = 0; i < number_of_processes; i++)
        {
            // some process arrives during our execution, add to ready_queue
            if (processes[i].arrival_time > cpu_cycle_begin_state && processes[i].arrival_time <= cpu_cycle_end_state)
            {
                ready_queue_counter++;
                ready_queue[ready_queue_counter] = i;
                process_counter++;
            }
        }

        // Check if the current process is done executing its burst cycles, if not then add it to ready queue as well
        if (burst_times[ready_queue[current_node]] > 0)
        {
            ready_queue_counter++;
            ready_queue[ready_queue_counter] = ready_queue[current_node];
        }

        // If its done, then calculate everything related to that processes.
        else if (burst_times[ready_queue[current_node]] == 0)
        {
            processes[ready_queue[current_node]].completion_time = cpu_cycle_end_state;
            processes[ready_queue[current_node]].turnaround_time = processes[ready_queue[current_node]].completion_time -
                                                                   processes[ready_queue[current_node]].arrival_time;
            processes[ready_queue[current_node]].waiting_time = processes[ready_queue[current_node]].turnaround_time -
                                                                processes[ready_queue[current_node]].burst_time;
        }

        // finally increment the current node to point to the next part of the ready queue counter
        current_node++;
    }
    // Print the last line of gantt chart
    printf("|%d|", cpu_time);
    printf("\n");
    for (int i = 0; i < ready_queue_size; i++)
        printf("-------------------");
    printf("\n\n");
}
```

```c
float calc_average_waiting_time(struct Process processes[], float
    number_of_processes)
{
    float a = 0;
    for (int i = 0; i < number_of_processes; i++)
    {
        a += processes[i].waiting_time;
    }
    printf("%d", a);

    a /= number_of_processes;
    printf("%f", a);

    return a;
}

float calc_average_tat(struct Process processes[], float number_of_processes)
{
    float a = 0;
    for (int i = 0; i < number_of_processes; i++)
    {
        a += processes[i].turnaround_time;
    }
    a /= number_of_processes;
    return a;
}

int main()
{
    int number_of_processes = 0, time_quantum = 2;
    float average_waiting_time, average_tat;
    // printf("How many Processes do you wanna input? \n\n");
    // scanf("%d", &number_of_processes);

    // if (number_of_processes == 0)
    // {
    //     printf("You do not have any processes\n");
    //     return 0;
    // }
    number_of_processes = 4;
    struct Process processes[4] =
        {
            {1, 7, 0, 0, 0, 0, 0},
            {2, 4, 2, 0, 0, 0, 0},
            {3, 1, 4, 0, 0, 0, 0},
            {4, 4, 5, 0, 0, 0, 0}

            // {1, 8, 0, 0, 0, 0, 0},
            // {2, 2, 5, 0, 0, 0, 0},
            // {3, 7, 1, 0, 0, 0, 0},
            // {4, 3, 6, 0, 0, 0, 0},
            // {5, 5, 8, 0, 0, 0, 0}
        };

    // accept_array(processes, number_of_processes);
    insertion_sort(processes, number_of_processes); // sort arrival times.

    // Display Round Robin

```

```
    printf("Here is the Gantt Chart for Round Robin Scheduling done on the Given Data:\n\n");
    round_robin(processes, number_of_processes, time_quantum);
    display(processes, number_of_processes, 0);
    average_waiting_time = calc_average_waiting_time(processes, number_of_processes);
    average_tat = calc_average_tat(processes, number_of_processes);
    printf("\nAverage waiting time is: %f", average_waiting_time);
    printf("\nAverage Turnaround time is: %f\n", average_tat);

    printf("\n\n Here is the Gantt Chart for Non Preemptive First Come First Server Scheduling done on the Given Data:\n\n");

    npfcfs(processes, number_of_processes);
    displayGanttChartFSFS(processes, number_of_processes);
    display(processes, number_of_processes, 0);
    average_waiting_time = calc_average_waiting_time(processes, number_of_processes);
    average_tat = calc_average_tat(processes, number_of_processes);
    printf("\nAverage waiting time is: %f", average_waiting_time);
    printf("\nAverage Turnaround time is: %f\n", average_tat);
}
```

Listing 1: Assignment 3.Cpp

2 Input and Output

```
Input Values In the order:
    int process_id;
    int burst_time;
    int arrival_time;
    int service_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
    int time_quantum;

{1, 7, 0, 0, 0, 0, 0}
{2, 4, 2, 0, 0, 0, 0}
{3, 1, 4, 0, 0, 0, 0}
{4, 4, 5, 0, 0, 0, 0}


Here is the Gantt Chart for Round Robin Scheduling done on the Given Data:

-----------------------------------------------------------------------------------------------
|0|      1        |2|      2        |4|      1        |6|      3        |7|      2        |9|      4        |11|     1        |13|     4        |15|     1        |16|
-----------------------------------------------------------------------------------------------


-----------------------------------------------------------------------------------------------
|        ID|      Bur|     Arr|     ser|     wait|    Com|     TAT
|         1|        7|       0|       0|        9|     16|      16|
|         2|        4|       2|       0|        3|      9|       7|
|         3|        1|       4|       0|        2|      7|       3|
|         4|        4|       5|       0|        6|     15|      10|
```

```
-----------------------------------------------------------------------------------------------------------
15.000000
Average waiting time is: 5.000000
Average Turnaround time is: 9.000000


 Here is the Gantt Chart for Non Preemptive First Come First Server Scheduling
    done on the Given Data:

================================================================
|       P1      ||      P2      ||      P3      ||      P4      |
----------------------------------------------------------------
|0|             |7|             |11|            |12|            |16|
================================================================
-----------------------------------------------------------------------------------------------------------
|       ID|     Bur|     Arr|     ser|     wait|    Com|     TAT
|        1|       7|       0|       0|        0|      7|       7|
|        2|       4|       2|       7|        5|     11|       9|
|        3|       1|       4|      11|        7|     12|       8|
|        4|       4|       5|      12|        7|     16|      11|
-----------------------------------------------------------------------------------------------------------
14.750000
Average waiting time is: 4.750000
Average Turnaround time is: 8.750000
```

Listing 2: Input and Output.Cpp

11/10/20

OS ASSIGNMENT - 3

Krishnaraj PT.
10322108888
PA20 ; (A 1)

Q.1. Explain the need for CPU scheduling.

→ Scheduling of processes / work is done to finish the work on time. CPU scheduling is a process that allows one process to use the CPU while another process is delayed (in standby), due to unavailability of any resources such as I/O etc. thus making full use of the CPU.

The purpose the CPU scheduling is to make the system more efficient, faster and fairer.

→ Whenever the ~~OS Operating System~~ CPU becomes idle, OS must ~~se~~ select one of the processes ready for launch, and assigns the CPU to one of them.

→ The selection process is done by a temporary CPU scheduler; the scheduler selects between memory process ready to launch, and assigns the CPU to one of them.

Q.2. Explain ~~for~~ Pre-emption and pre-emp. decision mode.

→ ① Non-Preemption : If the CPU is allocated to the process, then it keeps the CPU until it releases it by terminating or switching to waiting state.

② Pre-emptive : If CPU is allocated to a process, it may be released if high priority process needs the CPU.

Q.3 Explain FCFS and Round Robin with examples.

→ 1. FCFS : First come First serve.

→ Decision mode : Non-Preemptive.

→ Selection Function : max (t) — selects the process that is waiting for the maximum time.

→ Response Times : May be very high, especially if there is a large variance in the process execution times.

→ Overhead : Minimum.

→ Effect on Process : Penalizes short processes.

→ Starvation : NO.

Process	Arrival time	Burst time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

(*) **Gantt chart:**

P_1	P_2	P_3	P_4
0 – 7	7 – 11	11 – 12	12 – 16

(*) Avg. waiting time : $\dfrac{(0+5+7+7)}{4}$

$= \underline{4.75}$

(*) Avg. Turnaround Time $= \dfrac{(7+9+8+11)}{4}$

$= \underline{8.75}$

(2) **Round Robin:**

→ Each process gets a small amount of CPU time (time quantum)

→ After this time has elapsed, the process is preempted and added to the end of the ready time.

→ If there are n processes in the ready queue & the time quantum is q, then each process gets $1/n$ of the CPU in chunks of q units of time at once.

→ No process waits more than $(n-1)q$ time units

→ Selection function ~~Constant~~ : constant

→ Decision Mode : Preemption (at time quantum).

→ Response time : provides good response time for good ~~poor~~ short processes.

→ Overhead : Minimum.

eg. $q = 20$.

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

V. good

P_1	P_2	P_3	P_4	P_2	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 16

→ Avg. waiting time : $\frac{(81 + 20 + 94 + 97)}{4} = \underline{\underline{78}}$

→ Avg. Turnaround Time : $\frac{(134 + 37 + 162 + 121)}{4} = \underline{\underline{113.5}}$