Project: CPU SCHEDLUING ALGORITHMS

GROUP MEMBERS: Jeevanjot Singh, Atul Kumar, Rachit Sudan, Ananya Choudhary, Manmeet Kour, Amaan Khan.

Roll no's:
2021A1R063,2021A1R072,2021A1R083,2021A1R087, 2021A1R090,2022A1L004

Group Representative: Jeevanjot Singh.

Branch: Computer Science and Engineering

Semester: 3rd

ACKNOWLEDGEMENT

Through this section of our report, we want to present our gratitude towards our institute MIET. From here, we can work on projects that require you to analyse and solve a problem that exists in the real world. It is an experience that one can achieve rarely, and we are happy and thankful to get that chance here. We would also like to thank our mentor Dr. Swati Goel mam for being a guiding force throughout this.

Table of contents

Introduction

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Types of Scheduling: -

Pre-emptive Scheduling

In Pre-emptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority

task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Pre-emptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like pre-emptive.

Types of CPU scheduling Algorithm

There are mainly six types of process scheduling algorithms

1. First Come First Serve (FCFS)
2. Shortest-Job-First (SJF) Scheduling
3. Shortest Remaining Time
4. Priority Scheduling
5. Round Robin Scheduling
6. Multilevel Queue Scheduling

First Come First Serve

First Come First Serve is the full form of FCFS. It is the easiest and most simple CPU scheduling algorithm. In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

Shortest Remaining Time

The full form of SRT is Shortest remaining time. It is also known as SJF pre-emptive scheduling. In this method, the process will be allocated to the task, which is closest to its completion. This method prevents a newer ready state process from holding the completion of an older process.

Priority Based Scheduling

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler selects the tasks to work as per the priority.

Priority scheduling also helps OS to involve priority assignments. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority can be decided based on memory requirements, time requirements, etc.

Round-Robin Scheduling

Round robin is the oldest, simplest scheduling algorithm. The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turn. It is mostly used for scheduling algorithms in multitasking. This algorithm method helps for starvation free execution of processes.

Shortest Job First

SJF is a full form of (Shortest job first) is a scheduling algorithm in which the process with the shortest execution time should be selected for execution next. This scheduling method can be pre-emptive or non-pre-emptive. It significantly reduces the average waiting time for other processes awaiting execution.

Technical Detail

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
void fcfs();
void sjf();
void priority();
void srtf();
void roundrobin();
int findmax(int a, int b)
{
    return a > b ? a : b;
}

int findmin(int a, int b)
{
    return a < b ? a : b;
}
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void main()
{
    while (1)
    {
        printf("\n\n\t\t\tCPU SCHEDULING ALGORITHMS.");
        int choice;
        printf("\n1.Non-Premitive\t\t2.Premitive\t\t3.Exit\n");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
```

```
            printf("1.FCFS\t2.SJF\t3.PRIORTY\n");
            scanf("%d", &choice);
            switch (choice)
            {
            case 1:
                fcfs();
                break;
            case 2:
                sjf();
                break;
            case 3:
                priority();
                break;
            default:
                printf("Invalid input");
            }
            break;
        case 2:
        {
            printf("\n1.SRTF\t2.Round Robin\n");
            scanf("%d", &choice);
            switch (choice)
            {
            case 1:
                srtf();
                break;
            case 2:
                roundrobin();
                break;
            default:
                printf("Invalid input");
            }
            break;
        }
        case 3:
            exit(0);
        default:
            printf("Invalid input");
        }
    }
}
void priority()
{
    int n, avg_wt = 0;
    printf("Enter Number of Processes: ");
    scanf("%d", &n);

    // b is array for burst time, p for priority and index for process id
```

```
    int b[n], p[n], index[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter Burst Time and Priority Value for Process %d: ", i
+ 1);
        scanf("%d %d", &b[i], &p[i]);
        index[i] = i + 1;
    }
    for (int i = 0; i < n; i++)
    {
        int a = p[i], m = i;

        // Finding out highest priority element and placing it at its
desired position
        for (int j = i; j < n; j++)
        {
            if (p[j] > a)
            {
                a = p[j];
                m = j;
            }
        }

        // Swapping processes
        swap(&p[i], &p[m]);
        swap(&b[i], &b[m]);
        swap(&index[i], &index[m]);
    }

    // T stores the starting time of process
    int t = 0;

    // Printing scheduled process
    printf("Order of process Execution is\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d is executed from %d to %d\n", index[i], t, t + b[i]);
        t += b[i];
    }
    printf("\n");
    printf("Process Id     Burst Time   Wait Time    TurnAround Time\n");
    int wait_time = 0;
    for (int i = 0; i < n; i++)
    {
        printf("P%d             %d             %d             %d\n",
index[i], b[i], wait_time, wait_time + b[i]);
        avg_wt += wait_time;
        wait_time += b[i];
```

```
    }
    printf("Average waiting time is: %g\n", (avg_wt * 1.0) / n);
}
void sjf()
{
    int i, n, p[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, min, k = 1, btime
= 0;
    int bt[10], temp, j, at[10], wt[10], tt[10], ta = 0, sum = 0;
    float wavg = 0, tavg = 0, tsum = 0, wsum = 0;
    printf(" -------Shortest Job First Scheduling ( NP )-------\n");
    printf("\nEnter the No. of processes :");
    scanf("%d", &n);

    for (i = 0; i < n; i++)
    {
        printf("\tEnter the burst time of %d process :", i + 1);
        scanf(" %d", &bt[i]);
        printf("\tEnter the arrival time of %d process :", i + 1);
        scanf(" %d", &at[i]);
    }

    /*Sorting According to Arrival Time*/

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (at[i] < at[j])
            {
                temp = p[j];
                p[j] = p[i];
                p[i] = temp;
                temp = at[j];
                at[j] = at[i];
                at[i] = temp;
                temp = bt[j];
                bt[j] = bt[i];
                bt[i] = temp;
            }
        }
    }

    /*Arranging the table according to Burst time,
    Execution time and Arrival Time
    Arrival time <= Execution time
    */

    for (j = 0; j < n; j++)
```

```
    {
        btime = btime + bt[j];
        min = bt[k];
        for (i = k; i < n; i++)
        {
            if (btime >= at[i] && bt[i] < min)
            {
                temp = p[k];
                p[k] = p[i];
                p[i] = temp;
                temp = at[k];
                at[k] = at[i];
                at[i] = temp;
                temp = bt[k];
                bt[k] = bt[i];
                bt[i] = temp;
            }
        }
        k++;
    }
    wt[0] = 0;
    for (i = 1; i < n; i++)
    {
        sum = sum + bt[i - 1];
        wt[i] = sum - at[i];
        wsum = wsum + wt[i];
    }

    wavg = (wsum / n);
    for (i = 0; i < n; i++)
    {
        ta = ta + bt[i];
        tt[i] = ta - at[i];
        tsum = tsum + tt[i];
    }

    tavg = (tsum / n);

    printf("************************");
    printf("\n RESULT:-");
    printf("\nProcess\t Burst\t Arrival\t Waiting\t Turn-around");
    for (i = 0; i < n; i++)
    {
        printf("\n p%d\t %d\t %d\t\t%d\t\t%d", p[i], bt[i], at[i], wt[i],
tt[i]);
    }

    printf("\n\nAVERAGE WAITING TIME : %.2f", wavg);
```

```
    printf("\nAVERAGE TURN AROUND TIME : %.2f", tavg);
}
void fcfs()
{
    int n, bt[20], wt[20], tat[20], avwt = 0, avtat = 0, i, j;
    printf("Enter total number of processes(maximum 20):");
    scanf("%d", &n);

    printf("Enter Process Burst Timen");
    for (i = 0; i < n; i++)
    {
        printf("P[%d]:", i + 1);
        scanf("%d", &bt[i]);
    }

    wt[0] = 0;

    for (i = 1; i < n; i++)
    {
        wt[i] = 0;
        for (j = 0; j < i; j++)
            wt[i] += bt[j];
    }

    printf("ProcessttBurst TimetWaiting TimetTurnaround Completion
Time\n");

    for (i = 0; i < n; i++)
    {
        tat[i] = bt[i] + wt[i];
        avwt += wt[i];
        avtat += tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i],
tat[i]);
    }

    avwt /= i;
    avtat /= i;
    printf("\n\nAverage Waiting Time:%d\n", avwt);
    printf("\nAverage Turnaround Time:%d", avtat);
}
void srtf()
{
    struct process_struct
    {
        int pid;
        int at;
        int bt;
```

```
        int ct, wt, tat, rt, start_time;
    } ps[100];

    int n;
    float bt_remaining[100];
    bool is_completed[100] = {false}, is_first_process = true;
    int current_time = 0;
    int completed = 0;
    ;
    float sum_tat = 0, sum_wt = 0, sum_rt = 0, total_idle_time = 0, prev
= 0;

    int max_completion_time, min_arrival_time;

    printf("Enter total number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        printf("\nEnter Process %d Arrival Time: ", i);
        scanf("%d", &ps[i].at);
        ps[i].pid = i;
    }

    for (int i = 0; i < n; i++)
    {
        printf("\nEnter Process %d Burst Time: ", i);
        scanf("%d", &ps[i].bt);
        bt_remaining[i] = ps[i].bt;
    }

    while (completed != n)
    {
        // find process with min. burst time in ready queue at current
time
        int min_index = -1;
        int minimum = INT_MAX;
        for (int i = 0; i < n; i++)
        {
            if (ps[i].at <= current_time && is_completed[i] == false)
            {
                if (bt_remaining[i] < minimum)
                {
                    minimum = bt_remaining[i];
                    min_index = i;
                }
                if (bt_remaining[i] == minimum)
                {
                    if (ps[i].at < ps[min_index].at)
```

```
                    {
                        minimum = bt_remaining[i];
                        min_index = i;
                    }
                }
            }
        }

        if (min_index == -1)
        {
            current_time++;
        }
        else
        {
            if (bt_remaining[min_index] == ps[min_index].bt)
            {
                ps[min_index].start_time = current_time;
                total_idle_time += (is_first_process == true) ? 0 :
(ps[min_index].start_time - prev);
                is_first_process = false;
            }
            bt_remaining[min_index] -= 1;
            current_time++;
            prev = current_time;
            if (bt_remaining[min_index] == 0)
            {
                ps[min_index].ct = current_time;
                ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
                ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
                ps[min_index].rt = ps[min_index].start_time -
ps[min_index].at;

                sum_tat += ps[min_index].tat;
                sum_wt += ps[min_index].wt;
                sum_rt += ps[min_index].rt;
                completed++;
                is_completed[min_index] = true;
                // total_idle_time += (is_first_process==true) ? 0 :
(ps[min_index].start_time -  prev);
                // prev= ps[min_index].ct; // or current_time;
            }
        }
    }
    // Calculate Length of Process completion cycle
    max_completion_time = INT_MIN;
    min_arrival_time = INT_MAX;
    for (int i = 0; i < n; i++)
    {
```

```
        max_completion_time = findmax(max_completion_time, ps[i].ct);
        min_arrival_time = findmin(min_arrival_time, ps[i].at);
    }

    // Output
    printf("\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t%d\t\t%d\t%d\t%d\t%d\n", ps[i].pid, ps[i].at,
ps[i].bt, ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

    printf("\n");

    printf("\nAverage Turn Around time = %f ", (float)sum_tat / n);
    printf("\nAverage Waiting Time = %f ", (float)sum_wt / n);
    printf("\nAverage Response Tim e= %f ", (float)sum_rt / n);
}
void roundrobin()
{
    int n, i, qt, count = 0, temp, sq = 0, bt[10], wt[10], tat[10],
rem_bt[10];
    float awt = 0, atat = 0;

    printf("ENTER NUMBER OF PROCESS : ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("ENTER BURST TIME OF PROCESS :%d :  ", i + 1);
        scanf("%d", &bt[i]);
        rem_bt[i] = bt[i];
    }
    printf("ENTER QUANTUM TIME : ");
    scanf("%d", &qt);
    while (1)
    {
        for (i = 0, count = 0; i < n; i++)
        {
            temp = qt;
            if (rem_bt[i] == 0)
            {
                count++;
                continue;
            }
            if (rem_bt[i] > qt)
                rem_bt[i] = rem_bt[i] - qt;
            else if (rem_bt[i] >= 0)
            {
                temp = rem_bt[i];
                rem_bt[i] = 0;
```

```
            }
            sq = sq + temp;
            tat[i] = sq;
        }
        if (n == count)
            break;
    }
    printf("\nPROCESS\tBURST TIME\tTURN AROUND TIME\tWAITING TIME\n");
    for (i = 0; i < n; i++)
    {
        wt[i] = tat[i] - bt[i];
        awt = awt + wt[i];
        atat = atat + tat[i];
        printf("\n%d\t%d\t\t%d\t\t\t%d", i + 1, bt[i], tat[i], wt[i]);
    }
    awt = awt / n;
    atat = atat / n;
    printf("\nAVERAGE WAITING TIME=%f\n", awt);
    printf("AVERAGE TURN AROUND TIME=%f", atat);
}
```