1. Problem Statement

The major task of an operating system is to manage a collection of processes which is known as CPU scheduling. In this project, we have used three types of scheduling, i.e, FCFS(First Come First Serve), RM Scheduling (Rate Monotonic), and RR Scheduling (Round Robin). Here, we will be analyzing these scheduling methods to get which scheduling algorithm leads to maximum optimization.

2. Introduction

2.1 Motivation

Scheduling is a fundamental operating-system function. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler. The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them. CPU scheduling is the process of deciding which process will own the CPU to use while another process is suspended. Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, scheduling is central to the operating-system design. CPU scheduling determines which processes run when there are multiple run-able processes. CPU scheduling is important because it can have a big effect on resource utilization and the overall performance of the system. Therefore, knowing how processes are executed in our systems, the various types of scheduling algorithms that are available and do they affect overall throughput is important.

2.2 Objective

The primary objective is to show how different CPU scheduling algorithms work in the system and how they can affect resource utilization and the overall performance of the system. The goal of CPU scheduling is to ensure that as many jobs are running at a time as possible. In a single-CPU system, the goal is to keep one job running at all times. Multiprogramming allows us to keep many jobs ready to run at all times. In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it, and this procedure is called CPU scheduling. The Operating System uses various scheduling algorithms to schedule the processes. A scheduling algorithm is preemptive when a task running on a processor may be suspended in favor of a higher-priority task, then resume execution later. In the opposite case, it is a non-preemptive task.

2.3 Contribution

This Project will contribute to improve the overall performance of Scheduling and allocating resources equally to more deserving processes. It determines which processes run when there are multiple run-able processes and hence it can have a big effect on resource utilization and the overall performance of the system.

The primary objective of CPU scheduling is to ensure that as many jobs are running at a time as is possible. It will allow one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.

3. Detailed Description of Project

When the system has a choice of processes to execute, it must have a strategy called Process Scheduling.

These algorithms decide when and for how long each process runs; they make choices about

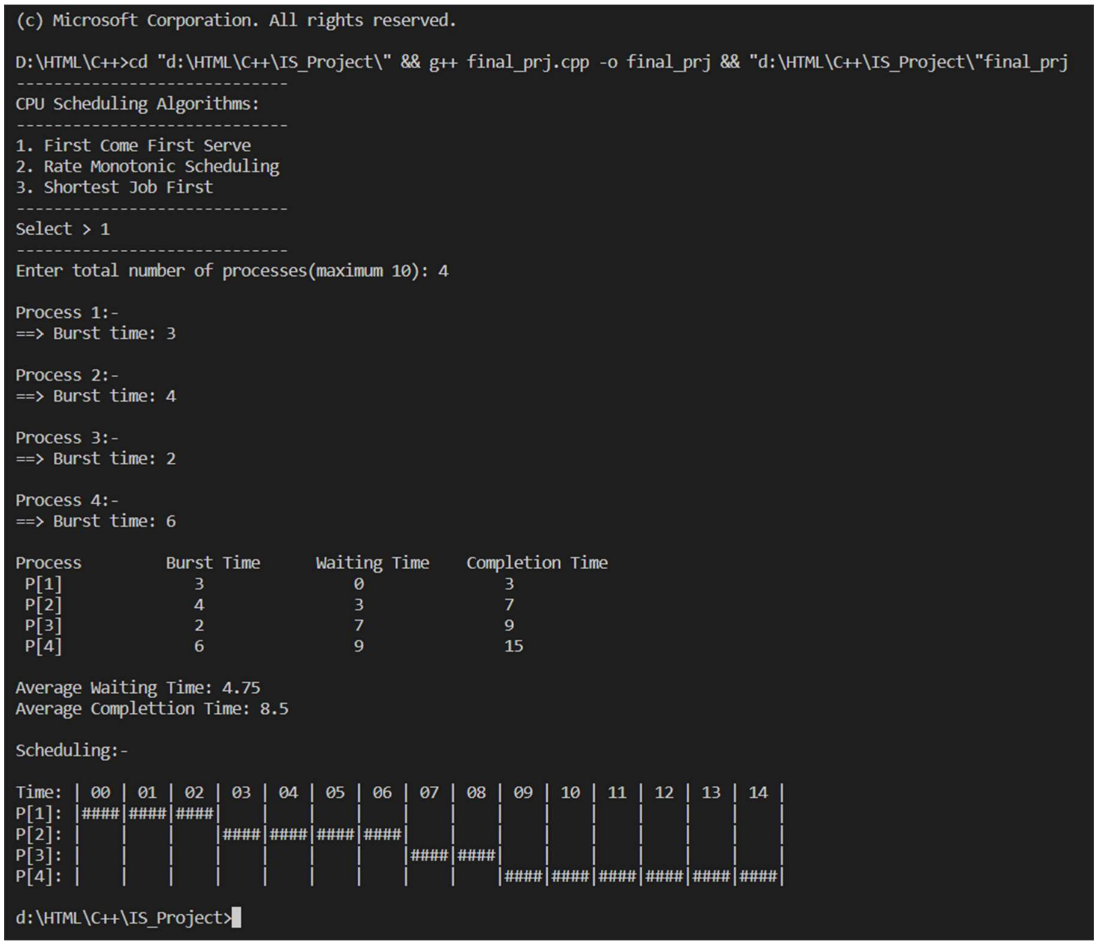
- Pre-emptive
- Priorities
- Running time
- Time-to-Completion
- Fairness

We will be simulating these Scheduling Algorithms and comparing them against various parameters mentioned above

4. Implementation

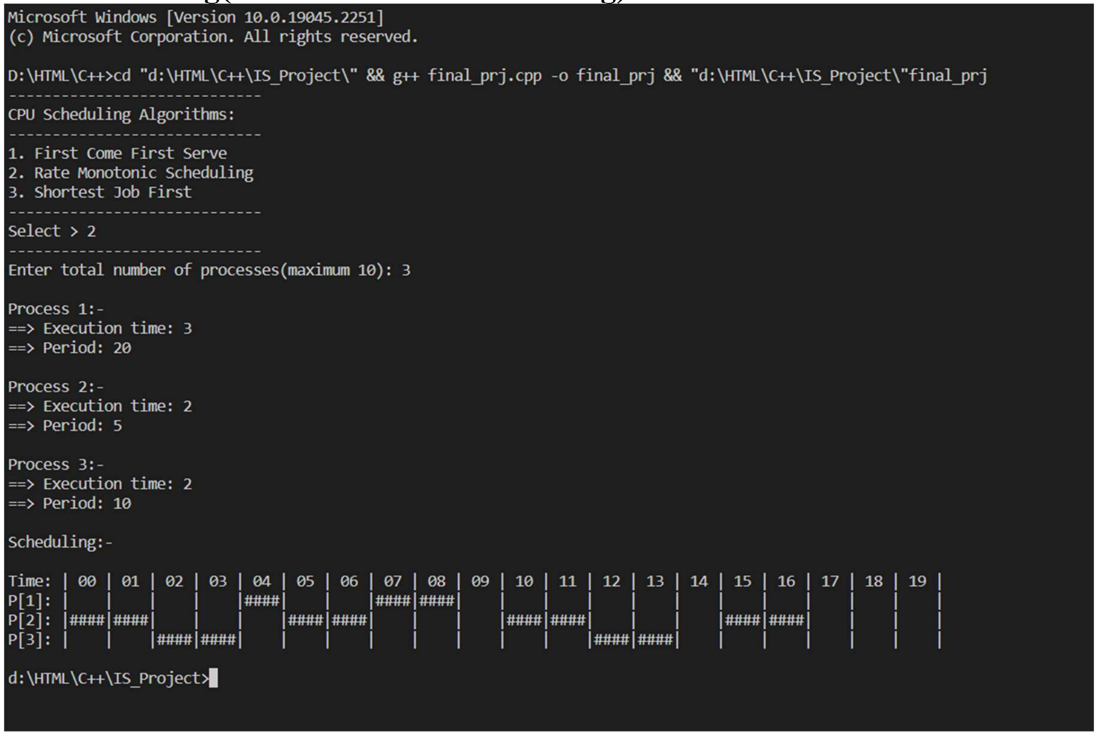
4.1 Snapshots of Simulation

FCFS (First Come First Serve)



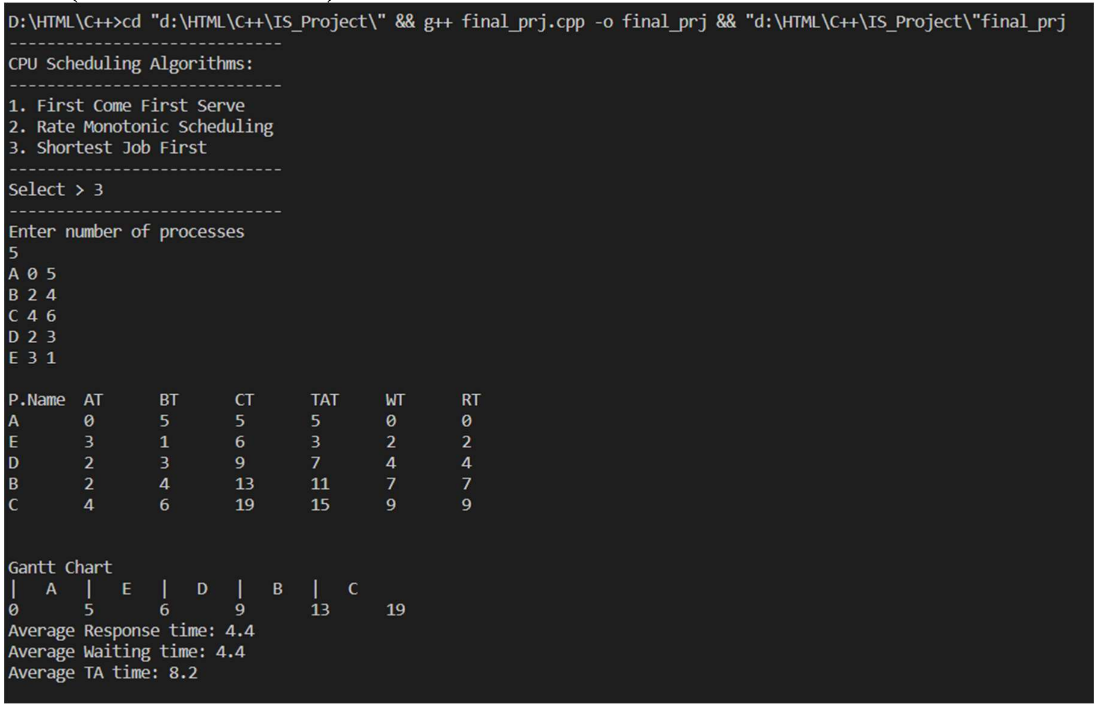
```
(c) Microsoft Corporation. All rights reserved.

D:\HTML\C++>cd "d:\HTML\C++\IS_Project\" && g++ final_prj.cpp -o final_prj && "d:\HTML\C++\IS_Project\"final_prj
-----------------------------
CPU Scheduling Algorithms:
-----------------------------
1. First Come First Serve
2. Rate Monotonic Scheduling
3. Shortest Job First
-----------------------------
Select > 1
-----------------------------
Enter total number of processes(maximum 10): 4

Process 1:-
==> Burst time: 3

Process 2:-
==> Burst time: 4

Process 3:-
==> Burst time: 2

Process 4:-
==> Burst time: 6

Process         Burst Time      Waiting Time    Completion Time
 P[1]              3               0               3
 P[2]              4               3               7
 P[3]              2               7               9
 P[4]              6               9               15

Average Waiting Time: 4.75
Average Complettion Time: 8.5

Scheduling:-

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 |
P[1]: |####|####|####|    |    |    |    |    |    |    |    |    |    |    |    |
P[2]: |    |    |    |####|####|####|####|    |    |    |    |    |    |    |    |
P[3]: |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |
P[4]: |    |    |    |    |    |    |    |    |    |####|####|####|####|####|####|

d:\HTML\C++\IS_Project>
```

RM Scheduling(Rate Monotonic Scheduling)



```
Microsoft Windows [Version 10.0.19045.2251]
(c) Microsoft Corporation. All rights reserved.

D:\HTML\C++>cd "d:\HTML\C++\IS_Project\" && g++ final_prj.cpp -o final_prj && "d:\HTML\C++\IS_Project\"final_prj
-----------------------------
CPU Scheduling Algorithms:
-----------------------------
1. First Come First Serve
2. Rate Monotonic Scheduling
3. Shortest Job First
-----------------------------
Select > 2
-----------------------------
Enter total number of processes(maximum 10): 3

Process 1:-
==> Execution time: 3
==> Period: 20

Process 2:-
==> Execution time: 2
==> Period: 5

Process 3:-
==> Execution time: 2
==> Period: 10

Scheduling:-

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
P[2]: |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
P[3]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |

d:\HTML\C++\IS_Project>
```

SJF (Shortest Job First)



```
D:\HTML\C++>cd "d:\HTML\C++\IS_Project\" && g++ final_prj.cpp -o final_prj && "d:\HTML\C++\IS_Project\"final_prj
-----------------------------
CPU Scheduling Algorithms:
-----------------------------
1. First Come First Serve
2. Rate Monotonic Scheduling
3. Shortest Job First
-----------------------------
Select > 3
-----------------------------
Enter number of processes
5
A 0 5
B 2 4
C 4 6
D 2 3
E 3 1

P.Name  AT      BT      CT      TAT     WT      RT
A       0       5       5       5       0       0
E       3       1       6       3       2       2
D       2       3       9       7       4       4
B       2       4       13      11      7       7
C       4       6       19      15      9       9


Gantt Chart
|   A   |   E   |   D   |   B   |   C
0       5       6       9       13      19
Average Response time: 4.4
Average Waiting time: 4.4
Average TA time: 8.2
```

4.2 Program Code

```
#include <bits/stdc++.h>
#include <numeric>
#include <math.h>
using namespace std;
#define MAX_PROCESS 10

int num_of_process = 3, count, remain, time_quantum;
int execution_time[MAX_PROCESS], period[MAX_PROCESS], remain_time[MAX_PROCESS],
deadline[MAX_PROCESS], remain_deadline[MAX_PROCESS];
int burst_time[MAX_PROCESS], wait_time[MAX_PROCESS],
completion_time[MAX_PROCESS], arrival_time[MAX_PROCESS];

struct node
{
    char pname[50];
    int btime;
    int atime;
} a[50];
void insert(int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        cin >> a[i].pname;
        cin >> a[i].atime;
        cin >> a[i].btime;
    }
}
bool btimeSort(node a, node b)
{
    return a.btime < b.btime;
}
bool atimeSort(node a, node b)
{
    return a.atime < b.atime;
}
void disp(int n)
{
    sort(a, a + n, btimeSort);
    sort(a, a + n, atimeSort);
    int ttime = 0, i;
    int j, tArray[n];
    for (i = 0; i < n; i++)
    {
        j = i;
```

```
            while (a[j].atime <= ttime && j != n)
            {
                j++;
            }
            sort(a + i, a + j, btimeSort);
            tArray[i] = ttime;
            ttime += a[i].btime;
        }
        tArray[i] = ttime;
        float averageWaitingTime = 0;
        float averageResponseTime = 0;
        float averageTAT = 0;
        cout << "\n";
        cout << "P.Name  AT\tBT\tCT\tTAT\tWT\tRT\n";
        for (i = 0; i < n; i++)
        {
            cout << a[i].pname << "\t";
            cout << a[i].atime << "\t";
            cout << a[i].btime << "\t";
            cout << tArray[i + 1] << "\t";
            cout << tArray[i] - a[i].atime + a[i].btime << "\t";
            averageTAT += tArray[i] - a[i].atime + a[i].btime;
            cout << tArray[i] - a[i].atime << "\t";
            averageWaitingTime += tArray[i] - a[i].atime;
            cout << tArray[i] - a[i].atime << "\t";
            averageResponseTime += tArray[i] - a[i].atime;
            cout << "\n";
        }
        cout << "\n";
        cout << "\nGantt Chart\n";
        for (i = 0; i < n; i++)
        {
            cout << "|   " << a[i].pname << "   ";
        }
        cout << "\n";
        for (i = 0; i < n + 1; i++)
        {
            cout << tArray[i] << "\t";
        }
        cout << "\n";
        cout << "Average Response time: " << (float)averageResponseTime / (float)n
    << endl;
        cout << "Average Waiting time: " << (float)averageWaitingTime / (float)n <<
    endl;
        cout << "Average TA time: " << (float)averageTAT / (float)n << endl;
    }

    // collecting details of processes
```

```
void get_process_info(int selected_algo)
{
    cout << "Enter total number of processes(maximum " << MAX_PROCESS << "): ";
    cin >> num_of_process;
    if (num_of_process < 1)
    {
        cout << "Do you really want to schedule " << num_of_process << "
processes? -_-" << endl;
        exit(0);
    }
    if (selected_algo == 2)
    {
        cout << endl
             << "Enter Time Quantum: ";
        cin >> time_quantum;
        if (time_quantum < 1)
        {
            cout << "Invalid Input: Time quantum should be greater than 0" <<
endl;
            exit(0);
        }
    }

    for (int i = 0; i < num_of_process; i++)
    {
        cout << endl
             << "Process " << i + 1 << ":-" << endl;
        if (selected_algo == 1)
        {
            cout << "==> Burst time: ";
            cin >> burst_time[i];
        }
        else if (selected_algo == 2)
        {
            cout << "=> Arrival Time: ";
            cin >> arrival_time[i];
            cout << "=> Burst Time: ";
            cin >> burst_time[i];
            remain_time[i] = burst_time[i];
        }
        else if (selected_algo > 2)
        {
            cout << "==> Execution time: ";
            cin >> execution_time[i];
            remain_time[i] = execution_time[i];
            if (selected_algo == 4)
            {
                cout << "==> Deadline: ";
                cin >> deadline[i];
            }
            else
            {
                cout << "==> Period: ";
                cin >> period[i];
            }
        }
    }
}

// get maximum of three numbers
int max(int a, int b, int c)
{
    long max, lcom, count, flag = 0;
    if (a >= b && a >= c)
        return max = a;
    else if (b >= a && b >= c)
        return max = b;
    else if (c >= a && c >= b)
        return max = c;
}

// calculating the observation time for scheduling timeline
int get_observation_time(int selected_algo)
{
    if (selected_algo < 3)
    {
        int sum = 0;
        for (int i = 0; i < num_of_process; i++)
        {
            sum += burst_time[i];
        }
        return sum;
    }

    else if (selected_algo == 3)
    {
        return max(period[0], period[1], period[2]);
    }

    else if (selected_algo == 4)
    {
        return max(deadline[0], deadline[1], deadline[2]);
    }
}

// print scheduling sequence
```

```
void print_schedule(int process_list[], int cycles)
{
    cout << endl
         << "Scheduling:-" << endl
         << endl;
    cout << "Time: ";
    for (int i = 0; i < cycles; i++)
    {
        if (i < 10)
            cout << "| 0" << i << " ";
        else
            cout << "| " << i << " ";
    }
    cout << "|" << endl;

    for (int i = 0; i < num_of_process; i++)
    {
        cout << "P[" << i + 1 << "]: ";
        for (int j = 0; j < cycles; j++)
        {
            if (process_list[j] == i + 1)
                cout << "|####";
            else
                cout << "|    ";
        }
        cout << "|" << endl;
    }
}

void first_come_first_serve(int time)
{
    int process_list[time];
    int execution_time[num_of_process];

    int accsum, total_wait_time = 0, total_completion_time = 0;
    float average_wait_time = 0.0, average_completion_time = 0.0;

    /* start computing waiting time */
    wait_time[0] = 0; // first process doesn't wait
    for (int i = 1; i < num_of_process; i++)
    {
        wait_time[i] = 0;
        for (int j = 0; j < i; j++)
            wait_time[i] += burst_time[j];
    } /* end computing waiting time */

    // computing completion time of each process
    partial_sum(burst_time, burst_time + num_of_process, completion_time);
    cout << endl
         << "Process\t\tBurst Time\tWaiting Time\tCompletion Time";

    for (int i = 0; i < num_of_process; i++)
    {
        accsum = burst_time[i];
        total_wait_time += wait_time[i];
        total_completion_time += completion_time[i];
        cout << endl
             << " P[" << i + 1 << "]\t\t   " << burst_time[i] << "\t\t    " <<
wait_time[i] << "\t\t    " << completion_time[i];
    }

    average_wait_time = total_wait_time * 1.0 / num_of_process;
    average_completion_time = total_completion_time * 1.0 / num_of_process;
    cout << endl
         << endl
         << "Average Waiting Time: " << average_wait_time;
    cout << endl
         << "Average Complettion Time: " << average_completion_time << endl;

    int proc = 0;
    // computing process list
    for (int i = 0; i < time; i++)
    {
        if (burst_time[proc]-- < 1)
        {
            proc++;
            burst_time[proc]--;
        }
        process_list[i] = proc + 1; // process' number start from 1 not 0.
    }

    print_schedule(process_list, time);
}

void rate_monotonic(int time)
{
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    {
        utilization += (1.0 * execution_time[i]) / period[i];
    }
    int n = num_of_process;
    if (utilization > n * (pow(2, 1.0 / n) - 1))
    {
        cout << endl
```

```
             << "Given problem is not schedulable under said scheduling
algorithm." << endl;
        exit(0);
    }

    int process_list[time] = {0}, min = 999, next_process = 0;
    for (int i = 0; i < time; i++)
    {
        min = 1000;
        for (int j = 0; j < num_of_process; j++)
        {
            if (remain_time[j] > 0)
            {
                if (min > period[j])
                {
                    min = period[j];
                    next_process = j;
                }
            }
        }

        if (remain_time[next_process] > 0)
        {
            process_list[i] = next_process + 1; // +1 for catering 0 array
index.
            remain_time[next_process] -= 1;
        }

        for (int k = 0; k < num_of_process; k++)
        {
            if ((i + 1) % period[k] == 0)
            {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
}

void earliest_deadline_first(int time)
{
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    {
        utilization += (1.0 * execution_time[i]) / deadline[i];
    }
    int n = num_of_process;
    if (utilization > 1)
    {
        cout << endl
             << "Given problem is not schedulable under said scheduling
algorithm." << endl;
        exit(0);
    }

    int process[num_of_process];
    int max_deadline, current_process = 0, min_deadline, process_list[time];
    ;
    bool is_ready[num_of_process];

    for (int i = 0; i < num_of_process; i++)
    {
        is_ready[i] = true;
        process[i] = i + 1;
    }

    max_deadline = deadline[0];
    for (int i = 1; i < num_of_process; i++)
    {
        if (deadline[i] > max_deadline)
            max_deadline = deadline[i];
    }

    for (int i = 0; i < num_of_process; i++)
    {
        for (int j = i + 1; j < num_of_process; j++)
        {
            if (deadline[j] < deadline[i])
            {
                int temp = execution_time[j];
                execution_time[j] = execution_time[i];
                execution_time[i] = temp;
                temp = deadline[j];
                deadline[j] = deadline[i];
                deadline[i] = temp;
                temp = process[j];
                process[j] = process[i];
                process[i] = temp;
            }
        }
    }

    for (int i = 0; i < num_of_process; i++)
    {
        remain_time[i] = execution_time[i];
```

```
            remain_deadline[i] = deadline[i];
        }

        for (int t = 0; t < time; t++)
        {
            if (current_process != -1)
            {
                --execution_time[current_process];
                process_list[t] = process[current_process];
            }
            else
                process_list[t] = 0;

            for (int i = 0; i < num_of_process; i++)
            {
                --deadline[i];
                if ((execution_time[i] == 0) && is_ready[i])
                {
                    deadline[i] += remain_deadline[i];
                    is_ready[i] = false;
                }
                if ((deadline[i] <= remain_deadline[i]) && (is_ready[i] == false))
                {
                    execution_time[i] = remain_time[i];
                    is_ready[i] = true;
                }
            }

            min_deadline = max_deadline;
            current_process = -1;
            for (int i = 0; i < num_of_process; i++)
            {
                if ((deadline[i] <= min_deadline) && (execution_time[i] > 0))
                {
                    current_process = i;
                    min_deadline = deadline[i];
                }
            }
        }
        print_schedule(process_list, time);
    }

    int main(int argc, char *argv[])
    {
        int option = 0;
        cout << "-----------------------------" << endl;
        cout << "CPU Scheduling Algorithms: " << endl;
        cout << "-----------------------------" << endl;
        cout << "1. First Come First Serve" << endl;
        cout << "2. Rate Monotonic Scheduling" << endl;
        cout << "3. Shortest Job First" << endl;
        cout << "----------------------------" << endl;
        cout << "Select > ";
        cin >> option;
        cout << "----------------------------" << endl;

        // get_process_info(option); // collecting processes detail
        // int observation_time = get_observation_time(option);

        if (option == 1)
        {
            get_process_info(option); // collecting processes detail
            int observation_time = get_observation_time(option);
            first_come_first_serve(observation_time);
        }

        // Rate Monotonic
        // 3 20, 2 5 , 2 10

        else if (option == 2)
        {
            get_process_info(3); // collecting processes detail
            int observation_time = get_observation_time(3);
            rate_monotonic(observation_time);
        }

        else if (option == 3)
        {
            int nop, choice, i;
            cout << "Enter number of processes\n";
            cin >> nop;
            insert(nop);
            disp(nop);
        }
        return 0;
    }
```

5. Results

- From the comparison of the results that are recorded, we can say that Shortest Job First has lots of advantages over other CPU scheduling algorithms namely First Come First Serve and Rate Monotonic Scheduling.
- The major advantage of this algorithm is that it gives the maximum throughput and minimum waiting time for a given set of processes and thus reduces the average waiting time.
- However, in this algorithm longer processes may never be processed by the system and may remain in the queue for very long time leading to starvation of processes which leads to lower CPU Utilization lower efficiency.
- However, FCFS algorithm is the easiest one to implement in any system because SJF requires knowing the length of the next CPU burst or request.
- Similar difficulty is observed in Rate Monotonic Scheduling where the priority is decided according to the cycle time of the processes involved, knowing which beforehand is a challenge in real life scenarios.
- Non-preemptive nature of algorithms leads to starvation of processes because other processes at the end of the queue have to wait till the process which is currently using CPU executes completely. Thus, there is a high average waiting time. They are easy to implement but poor in performance
- Preemptive algorithms do not stick to one single process. They keep on executing all processes for some duration continuously.

6. Conclusion

The project has successfully demonstrated the working of various preemptive and non-pre-emptive CPU scheduling algorithms. For fixed priority scheduling algorithms, due to its minimum average waiting time and average turnaround time, SJF scheduling algorithm serves all types of job with optimum scheduling criteria. But long process will never be served. Concerning dynamic priority scheduling algorithms, like Rate Monotonic Scheduling have a big advantage that they overcome the utilization limitations. The project will contribute to improve the overall performance of scheduling and allocating resources to more deserving processes and hence it will have a major impact on the resource utilization and overall performance of the system. It will allow one process to use the CPU while other process(es) is waiting for I/O, thereby achieving maximum utilization of CPU cycles

7. References

[1] SILBERSCHATZ, A. et P.B. GALVIN,Operating System Concepts. 8th Edition,Addison Wesley
.
[2] N. Goel, R.B. Garg “ A Comparative Study of CPU Scheduling Algorithms’’ , International Journal of Graphics & Image Processing |Vol 2|issue 4|November 2012.

[3] C. L. Liu and J W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environnement.

[4] J. Xu.Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. IEEE Trans. Softw. Eng., 19:139–154, February 1993.

[5] R. Naik, R.R.Manthalkar, Instantaneous Utilization Based Scheduling Algorithms for Real Time Systems, Pune University, SGGS Nanded, RadhakrishnaNaik et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (2) , 2011, 654-662.