**Batch: D - 1 Roll No.: 16010122096**

**Experiment No. 05**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **TITLE:** Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority ) |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** To implement basic Process management algorithms ( Round Robin,SRTN, Priority)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 2.** To understand the concept of process, thread and resource management.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems” McGraw Hill Third**

**Edition.**

1. **William Stallings, “Operating System Internal & Design Principles”, Pearson.**
2. **Andrew S. Tanenbaum, “Modern Operating System”, Prentice Hall.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Most systems handle numerous processes with short CPU bursts interspersed with I/O requests and a few processes with long CPU bursts. To ensure good time-sharing performance, a running process may be preempted to allow another to run. The ready list, or run queue, maintains all processes ready to run and not blocked by I/O or other system requests. Entries in this list point to the process control block, which stores all process information and state.

When an I/O request completes, the process moves from the waiting state to the ready state and is placed on the run queue. The process scheduler, a key component of the operating system, decides whether the current process should continue running or if another should take over. This decision is triggered by four events:

1. The current process issues an I/O request or system request, moving it from running to waiting.
2. The current process terminates.
3. A timer interrupt indicates the process has run for its allotted time, moving it from running to ready.
4. An I/O operation completes, moving the process from waiting to ready, potentially preempting the current process.

The scheduling algorithm, or policy, determines the sequence and duration of process execution, a complex task given the limited information about ready processes.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Description of the application to be implemented**:

# Shortest Remaining Time First Algorithm :

The SRTF algorithm is a preemptive version of the Shortest Job First (SJF) scheduling. It selects the process with the shortest remaining burst time to execute next. If a new process arrives with a shorter burst time than the currently running process, the current process is preempted, and the new process is executed. This method minimizes the average waiting time but may cause higher context switching, which can increase overhead. SRTF is particularly useful in time-sharing systems where responsiveness is important.

# Priority scheduling:

Priority scheduling assigns each process a priority, and the process with the highest priority is executed first. Priorities can be static (assigned once) or dynamic (changed over time). In some cases, higher numbers represent higher priorities, while in others, lower numbers indicate higher importance. The algorithm can be either preemptive, where a higher-priority process can preempt a running lower-priority one, or non-preemptive. This method allows more critical tasks to be executed quickly but can lead to starvation of lower-priority processes if not managed carefully.

**Implementation details:**

**SRTF:**

#include <bits/stdc++.h>

using namespace std;

void srtn(int n, vector<int> &arrival\_times, vector<int> &burst\_times)

{

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    vector<int> completion\_times(n), gantt;

    int time = 0, completed = 0;

    while (completed != n)

    {

        for (int i = 0; i < n; i++)

            if (arrival\_times[i] == time)

                pq.push({burst\_times[i], i});

        auto x = pq.top();

        pq.pop();

        x.first--;

        gantt.push\_back(x.second);

        if (x.first == 0)

        {

            completion\_times[x.second] = time + 1;

            completed++;

        }

        else

            pq.push(x);

        time++;

    }

    vector<int> turnaround\_times(n), waiting\_times(n);

    for (int i = 0; i < n; i++)

    {

        turnaround\_times[i] = completion\_times[i] - arrival\_times[i];

        waiting\_times[i] = turnaround\_times[i] - burst\_times[i];

    }

    int avt = accumulate(turnaround\_times.begin(), turnaround\_times.end(), 0);

    int avw = accumulate(waiting\_times.begin(), waiting\_times.end(), 0);

    vector<int> p, t;

    for (int i = 0; i < gantt.size(); i++)

    {

        if (i == 0 || gantt[i] != gantt[i - 1])

        {

            p.push\_back(gantt[i]);

            t.push\_back(i);

        }

    }

    t.push\_back(time);

    cout << "Completion Times: ";

    for (auto &i : completion\_times)

        cout << i << " ";

    cout << "\nTurnaround Times: ";

    for (auto &i : turnaround\_times)

        cout << i << " ";

    cout << "\nWaiting Times: ";

    for (auto &i : waiting\_times)

        cout << i << " ";

    cout << "\nAvg. Turnaround Time: " << (float)avt / n;

    cout << "\nAvg. Waiting Time: " << (float)avw / n;

    cout << "\nGantt Chart\n";

    for (auto &i : p)

        cout << " P" << i + 1 << " ";

    cout << "\n";

    for (auto &i : t)

        cout << i << "   ";

    cout << "\n";

}

int main()

{

    int n;

    cout << "Enter the number of processes: ";

    cin >> n;

    vector<int> burst\_times(n);

    vector<int> arrival\_times(n);

    cout << "Enter the arrival times: ";

    for (auto &i : arrival\_times)

        cin >> i;

    cout << "Enter the burst times: ";

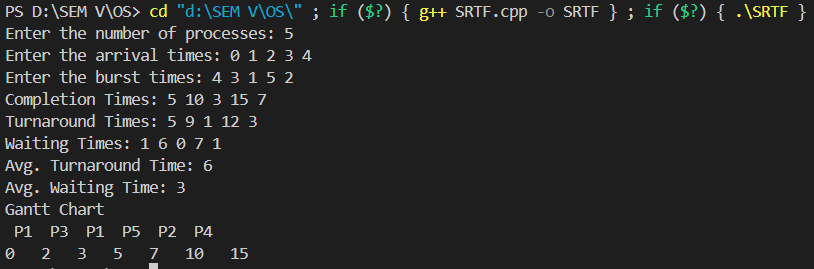
    for (auto &i : burst\_times)

        cin >> i;

    srtn(n, arrival\_times, burst\_times);

}

**Output:**

****

**PRIORITY:**

#include <bits/stdc++.h>

using namespace std;

void priority(int n, vector<int> arrival\_times, vector<int> burst\_times)

{

    priority\_queue<pair<int, int>> pq;

    vector<int> completion\_times(n), gantt;

    int time = 0, completed = 0;

    vector<int> priority(n);

    cout << "Enter the priority of the processes (Largest Number First): ";

    for (auto &i : priority)

        cin >> i;

    while (completed != n)

    {

        for (int i = 0; i < n; i++)

            if (arrival\_times[i] == time)

                pq.push({priority[i], i});

        auto x = pq.top();

        pq.pop();

        burst\_times[x.second]--;

        gantt.push\_back(x.second);

        if (burst\_times[x.second] == 0)

        {

            completion\_times[x.second] = time + 1;

            completed++;

        }

        else

            pq.push(x);

        time++;

    }

    vector<int> turnaround\_times(n), waiting\_times(n);

    for (int i = 0; i < n; i++)

    {

        turnaround\_times[i] = completion\_times[i] - arrival\_times[i];

        waiting\_times[i] = turnaround\_times[i] - burst\_times[i];

    }

    int avt = accumulate(turnaround\_times.begin(), turnaround\_times.end(), 0);

    int avw = accumulate(waiting\_times.begin(), waiting\_times.end(), 0);

    vector<int> p, t;

    for (int i = 0; i < gantt.size(); i++)

    {

        if (i == 0 || gantt[i] != gantt[i - 1])

        {

            p.push\_back(gantt[i]);

            t.push\_back(i);

        }

    }

    t.push\_back(time);

    cout << "Completion Times: ";

    for (auto &i : completion\_times)

        cout << i << " ";

    cout << "\nTurnaround Times: ";

    for (auto &i : turnaround\_times)

        cout << i << " ";

    cout << "\nWaiting Times: ";

    for (auto &i : waiting\_times)

        cout << i << " ";

    cout << "\nAvg. Turnaround Time: " << (float)avt / n;

    cout << "\nAvg. Waiting Time: " << (float)avw / n;

    cout << "\nGantt Chart\n";

    for (auto &i : p)

        cout << " P" << i + 1;

    cout << "\n";

    for (auto &i : t)

        cout << i << "  ";

    cout << "\n";

}

int main()

{

    int n;

    cout << "Enter the number of processes: ";

    cin >> n;

    vector<int> burst\_times(n);

    vector<int> arrival\_times(n);

    cout << "Enter the arrival times: ";

    for (auto &i : arrival\_times)

        cin >> i;

    cout << "Enter the burst times: ";

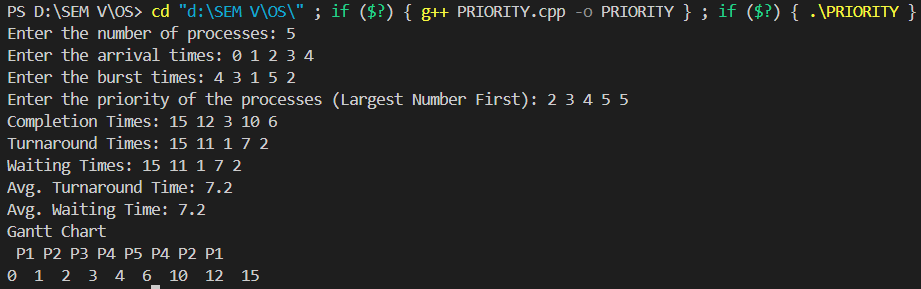
    for (auto &i : burst\_times)

        cin >> i;

    priority(n, arrival\_times, burst\_times);

}

**Output:**

****

**Conclusion:**

This experiment demonstrated the implementation of SRTN and Priority scheduling algorithms, enhancing understanding of process scheduling, resource management, and their impact on process performance metrics.

**Post Lab Descriptive Questions**

1. Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?

**Ans:**

Shortest remaining time ( SRT ) scheduling algorithm selects the process for execution which has the smallest amount of time remaining until completion.

Let three processes be p0, p1 and p2. Their execution time is 10, 20 and 30 respectively. p0 spends first 2 time units in I/O, 7 units of CPU time and finally 1 unit in I/O. p1 spends first 4 units in I/O, 14 units of CPU time and finally 2 units in I/O. p2 spends first 6 units in I/O, 21 units of CPU time and finally 3 units in I/O.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PID | AT | IO | BT | IO |
| P0 | 0 | 2 | 7 | 1 |
| P1 | 0 | 4 | 14 | 2 |
| P2 | 0 | 6 | 21 | 3 |

AT- Arrival Time, IO-input/output, BT-Burst Time

First process p0 will spend 2 units in IO, next 7 units in BT, then process p1 will spend 14 units in BT (as its 4 units of IO has been spent already when previous process was running) and ten process p2 will spend 21 units in BT (as its 6 units of IO has been spent already when previous processes were running) and at last 3 units in IO (process

p0,p1,p2’s last IO included.) idle p0 p1 p2 idle 0 2 9 23 44 47

* Total time spent = 47
* Idle time = 2 + 3 = 5
* Percentage of idle time = (5/47) \* 100 = 10.6 %

1. What effect the time quantum has on its performance. What are the advantages and disadvantages of using a small versus a large time quantum?

**Ans:**

* **Small Time Quantum:**
* **Advantages:** More responsive system, better interactivity for real-time tasks.
* **Disadvantages:** Increased context switching, which may lead to overhead and inefficiency.
* **Large Time Quantum:**
* **Advantages:** Less overhead from context switching, more efficient for CPU-bound processes.
* **Disadvantages:** Reduced responsiveness for interactive tasks, longer waiting time for short processes.

**Date: Signature of faculty in-charge**