K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch:	D-2	**Roll No.:**	16010122151

Experiment No. 04

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

Signature of the Staff In-charge with date

TITLE: Implementation of Basic Process management algorithms – Non Pre-emptive (FCFS , SJF, priority)

AIM: To implement basic Non –Pre-emptive Process management algorithms (FCFS , SJF , 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.**
3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **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:

First-Come, First-Served Scheduling:

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

bool compareArrival(Process a, Process b) {
    return a.arrival_time < b.arrival_time;
}

void calculateFCFS(std::vector<Process>& processes) {
    int current_time = 0;
```

```
    for (auto& process : processes) {
        if (current_time < process.arrival_time) {
            current_time = process.arrival_time; // Wait until the
process arrives
        }
        current_time += process.burst_time; // Process execution
        process.completion_time = current_time; // Set completion time
        process.turnaround_time = process.completion_time -
process.arrival_time; // Calculate turnaround time
        process.waiting_time = process.turnaround_time -
process.burst_time; // Calculate waiting time
    }
}

void printProcesses(const std::vector<Process>& processes) {
    std::cout << "Process ID | Arrival Time | Burst Time | Completion
Time | Turnaround Time | Waiting Time\n";
    std::cout << "-----------------------------------------------------
-----------------------------\n";
    for (const auto& process : processes) {
        std::cout << process.id << "\t\t"
                  << process.arrival_time << "\t\t"
                  << process.burst_time << "\t\t"
                  << process.completion_time << "\t\t\t"
                  << process.turnaround_time << "\t\t\t"
                  << process.waiting_time << "\n";
    }
}

void printGanttChart(const std::vector<Process>& processes) {
    std::cout << "\nGantt Chart:\n";

    // Print the top line of the Gantt chart
    for (const auto& process : processes) {
        std::cout << "| P" << process.id << " ";
    }
    std::cout << "|\n";

    // Print the timeline
    std::cout << "0"; // Starting time
    int current_time = 0;
    for (const auto& process : processes) {
        current_time += process.burst_time;
```

```
        std::cout << "    " << current_time; // Print the end time of
each process
    }
    std::cout << "\n";
}

int main() {
    std::vector<Process> processes = {
        {1, 0, 5},
        {2, 1, 3},
        {3, 2, 8},
        {4, 3, 6}
    };

    std::sort(processes.begin(), processes.end(), compareArrival);
    calculateFCFS(processes);
    printProcesses(processes);
    printGanttChart(processes);

    return 0;
}
```

OUTPUT:-

```
Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time
--------------------------------------------------------------------------------
1       0       5       5           5           0
2       1       3       8           7           4
3       2       8       16          14          6
4       3       6       22          19          13

Gantt Chart:
| P1 | P2 | P3 | P4 |
0    5    8    16    22


=== Code Execution Successful ===
```

Shortest job first :

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

bool compareBurstTime(Process a, Process b) {
    return a.burst_time < b.burst_time;
}

void calculateSJF(std::vector<Process>& processes) {
    int current_time = 0;
    int completed = 0;
    int n = processes.size();
    std::vector<bool> is_completed(n, false);

    while (completed < n) {
        // Find the next process to execute (shortest burst time)
        int idx = -1;
        for (int i = 0; i < n; ++i) {
            if (!is_completed[i]) {
                if (idx == -1 || processes[i].burst_time <
processes[idx].burst_time) {
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            current_time += processes[idx].burst_time;
            processes[idx].completion_time = current_time;
            processes[idx].turnaround_time =
processes[idx].completion_time - processes[idx].arrival_time;
```

```
            processes[idx].waiting_time = processes[idx].turnaround_time
- processes[idx].burst_time;
            is_completed[idx] = true;
            completed++;
        }
    }
}

void printProcesses(const std::vector<Process>& processes) {
    std::cout << "Process ID | Arrival Time | Burst Time | Completion
Time | Turnaround Time | Waiting Time\n";
    std::cout << "-----------------------------------------------------
------------------------------\n";
    for (const auto& process : processes) {
        std::cout << process.id << "\t\t"
                  << process.arrival_time << "\t\t"
                  << process.burst_time << "\t\t"
                  << process.completion_time << "\t\t\t"
                  << process.turnaround_time << "\t\t\t"
                  << process.waiting_time << "\n";
    }
}

void printGanttChart(const std::vector<Process>& processes) {
    std::cout << "\nGantt Chart:\n";

    // Print the top line of the Gantt chart
    for (const auto& process : processes) {
        std::cout << "| P" << process.id << " ";
    }
    std::cout << "|\n";

    // Print the timeline
    std::cout << "0"; // Starting time
    int current_time = 0;
    for (const auto& process : processes) {
        current_time += process.burst_time;
        std::cout << "    " << current_time; // Print the end time of
each process
    }
    std::cout << "\n";
}
```

```cpp
int main() {
    std::vector<Process> processes = {
        {1, 0, 8},
        {2, 0, 4},
        {3, 0, 9},
        {4, 0, 5}
    };

    // Sort by burst time only, since all arrive at the same time
    std::sort(processes.begin(), processes.end(), compareBurstTime);
    calculateSJF(processes);
    printProcesses(processes);
    printGanttChart(processes);

    return 0;
}
```

OUTPUT

```
Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time
--------------------------------------------------------------------------------
2       0       4       4           4           0
4       0       5       9           9           4
1       0       8       17          17          9
3       0       9       26          26          17

Gantt Chart:
| P2 | P4 | P1 | P3 |
0    4    9    17    26


=== Code Execution Successful ===
```

Priority scheduling

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int priority; // Lower value indicates higher priority
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

bool comparePriority(Process a, Process b) {
    return a.priority < b.priority; // Higher priority first
}

void calculatePriorityScheduling(std::vector<Process>& processes) {
    int current_time = 0;
    int completed = 0;
    int n = processes.size();
    std::vector<bool> is_completed(n, false);

    while (completed < n) {
        // Find the next process to execute based on priority
        int idx = -1;
        for (int i = 0; i < n; ++i) {
            if (!is_completed[i]) {
                if (idx == -1 || processes[i].priority <
processes[idx].priority) {
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            current_time += processes[idx].burst_time;
            processes[idx].completion_time = current_time;
            processes[idx].turnaround_time =
processes[idx].completion_time - processes[idx].arrival_time;
```

```
            processes[idx].waiting_time = processes[idx].turnaround_time
- processes[idx].burst_time;
            is_completed[idx] = true;
            completed++;
        }
    }
}

void printProcesses(const std::vector<Process>& processes) {
    std::cout << "Process ID | Arrival Time | Burst Time | Priority |
Completion Time | Turnaround Time | Waiting Time\n";
    std::cout << "-------------------------------------------------------
------------------------------\n";
    for (const auto& process : processes) {
        std::cout << process.id << "\t\t"
                  << process.arrival_time << "\t\t"
                  << process.burst_time << "\t\t"
                  << process.priority << "\t\t"
                  << process.completion_time << "\t\t\t"
                  << process.turnaround_time << "\t\t\t"
                  << process.waiting_time << "\n";
    }
}

void printGanttChart(const std::vector<Process>& processes) {
    std::cout << "\nGantt Chart:\n";

    // Print the top line of the Gantt chart
    for (const auto& process : processes) {
        std::cout << "| P" << process.id << " ";
    }
    std::cout << "|\n";

    // Print the timeline
    std::cout << "0"; // Starting time
    int current_time = 0;
    for (const auto& process : processes) {
        current_time += process.burst_time;
        std::cout << "    " << current_time; // Print the end time of
each process
    }
    std::cout << "\n";
}
```

```cpp
int main() {
    std::vector<Process> processes = {
        {1, 0, 8, 2}, // Process ID 1 with priority 2
        {2, 0, 4, 1}, // Process ID 2 with priority 1 (highest)
        {3, 0, 9, 4}, // Process ID 3 with priority 4
        {4, 0, 5, 3}  // Process ID 4 with priority 3
    };

    // Sort by priority only, since all arrive at the same time
    std::sort(processes.begin(), processes.end(), comparePriority);
    calculatePriorityScheduling(processes);
    printProcesses(processes);
    printGanttChart(processes);

    return 0;
}
```

OUTPUT:-

```
Process ID | Arrival Time | Burst Time | Priority | Completion Time | Turnaround Time | Waiting
    Time
-------------------------------------------------------------------------------
2       0       4       1       4               4               0
1       0       8       2       12              12              4
4       0       5       3       17              17              12
3       0       9       4       26              26              17

Gantt Chart:
| P2 | P1 | P4 | P3 |
0    4    12    17    26
```

Conclusion: Through this experiment we understood the concept of non-pre-emptive scheduling algorithm and implemented First Come First Serve and Shortest Job First algorithm in C++ language.

Post Lab Questions

1. What is a criterion to evaluate a scheduling algorithm?
   a. • **Throughput**: Number of completed processes per time unit.
   b. • **Turnaround Time**: Total time from submission to completion.
   c. • **Waiting Time**: Time a process spends waiting in the ready queue.
   d. • **Response Time**: Time from submission until the first response.
   e. • **Fairness**: Equal CPU time for all processes, avoiding starvation.
   f. • **Overhead**: Resources consumed by the scheduling algorithm itself.
   g. • **Scalability**: Performance as the number of processes increases.
   h. • **Adaptability**: Ability to adjust to workload changes.
2. Analyse the efficiency and suitability of FCFS, SJF, and Priority scheduling algorithms.

FCFS (First-Come, First-Served)

Efficiency:

- **Throughput**: Generally low, especially if longer processes arrive first (convoy effect).
- **Turnaround Time**: Can be high, as processes may wait for others to complete.
- **Waiting Time**: Often increases with longer processes ahead in the queue.

Suitability:

- **Best for**: Simple systems where tasks are roughly the same length.
- **Not suitable for**: Time-sharing or interactive systems, as it can lead to high waiting times and poor response for shorter tasks.

2. SJF (Shortest Job First)

Efficiency:

- **Throughput**: Higher than FCFS, as shorter processes are completed quickly.
- **Turnaround Time**: Generally lower, especially for short processes.
- **Waiting Time**: Tends to be reduced, as shorter jobs finish before longer ones.

Suitability:

- **Best for**: Batch systems where job lengths are predictable.

- **Not suitable for**: Real-time systems due to the risk of starvation for longer processes, and it requires knowledge of job lengths in advance.

3. Priority Scheduling

Efficiency:

- **Throughput**: Can vary depending on how priorities are assigned.
- **Turnaround Time**: Can be low for high-priority tasks but may increase for lower-priority tasks.
- **Waiting Time**: May lead to increased waiting time for lower-priority processes, especially if they are starved.

3. A brief explanation of the concept of "starvation" in SJF scheduling and how to avoid it.

Starvation occurs in scheduling algorithms like Shortest Job First (SJF) when longer processes are perpetually postponed because shorter processes keep arriving. Since SJF prioritizes the execution of the shortest tasks, a long job may wait indefinitely if shorter jobs continue to enter the queue. This can lead to a situation where the long job is never executed, resulting in increased waiting time and frustration.

How to Avoid Starvation

1. **Aging**: Implement an aging mechanism that gradually increases the priority of waiting processes over time. As a process waits longer, its effective priority increases, making it more likely to be executed sooner.
2. **Time Quotas**: Allow processes a minimum amount of CPU time after which they are scheduled again. This ensures that longer jobs get a chance to execute periodically.
3. **Hybrid Scheduling**: Combine SJF with other algorithms, such as Round Robin, where SJF is used for short tasks while longer tasks are not indefinitely delayed.
4. **Preemptive Scheduling**: If a new shorter job arrives, allow the scheduler to preempt the currently running longer job, but ensure that longer jobs get their share over time.

Date: 10-10-2024 **Signature of faculty in-charge**