

| **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.**

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**:

**First-Come, First-Served Scheduling:**

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve. As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

# Shortest job first :

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

There are basically two types of SJF methods:

* Non-Preemptive SJF
* Preemptive SJF

# Priority scheduling

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.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 depends upon memory requirements, time requirements, etc.

**Implementation details:** (printout of code)

FCFS:

#include <iostream>

#include <vector>

#include <iomanip>

#include <algorithm>

using namespace std;

struct Process {

int id;

int arrival;

int burst;

int waiting;

int turnaround;

int completion;

int start;

};

bool compareArrival(const Process& p1, const Process& p2) {

return p1.arrival < p2.arrival;

}

void calculateTimes(vector<Process>& processes) {

int n = processes.size();

int currentTime = 0;

for (int i = 0; i < n; i++) {

if (currentTime < processes[i].arrival) {

currentTime = processes[i].arrival;

}

processes[i].start = currentTime;

processes[i].completion = currentTime + processes[i].burst;

processes[i].turnaround = processes[i].completion - processes[i].arrival;

processes[i].waiting = processes[i].turnaround - processes[i].burst;

currentTime = processes[i].completion;

}

}

void findAvgTime(vector<Process>& processes) {

int n = processes.size();

calculateTimes(processes);

int total\_wt = 0, total\_tat = 0;

cout << setw(10) << "Process ID" << setw(15) << "Arrival Time"

<< setw(15) << "Burst Time" << setw(15) << "Waiting Time"

<< setw(20) << "Turnaround Time" << setw(15) << "Completion Time" << endl;

for (const auto& process : processes) {

total\_wt += process.waiting;

total\_tat += process.turnaround;

cout << setw(10) << process.id << setw(15) << process.arrival

<< setw(15) << process.burst << setw(15) << process.waiting

<< setw(20) << process.turnaround << setw(15) << process.completion << endl;

}

double avg\_wt = static\_cast<double>(total\_wt) / n;

double avg\_tat = static\_cast<double>(total\_tat) / n;

cout << fixed << setprecision(2);

cout << "\nAverage Waiting Time: " << avg\_wt << endl;

cout << "Average Turnaround Time: " << avg\_tat << endl;

}

void printGanttChart(const vector<Process>& processes) {

int n = processes.size();

int maxBurstTime = 0;

int maxCompletionTime = 0;

for (const auto& process : processes) {

if (process.burst > maxBurstTime) {

maxBurstTime = process.burst;

}

if (process.completion > maxCompletionTime) {

maxCompletionTime = process.completion;

}

}

int width = maxBurstTime + 3;

cout << "\nGantt Chart:" << endl;

cout << " ";

for (int i = 0; i < n; i++) {

cout << setw(width) << setfill('-') << " ";

}

cout << setfill(' ') << endl;

cout << " ";

for (int i = 0; i < n; i++) {

cout << setw(width) << processes[i].start << " ";

}

cout << setw(width) << processes.back().completion << endl;

cout << "|";

for (const auto& process : processes) {

cout << setw(width) << process.id << " |";

}

cout << endl;

cout << " ";

for (int i = 0; i < n; i++) {

cout << setw(width) << setfill('-') << " ";

}

cout << setfill(' ') << endl;

cout << " ";

for (int i = 0; i < n; i++) {

cout << setw(width) << processes[i].completion << " ";

}

cout << endl;

}

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

vector<Process> processes(n);

for (int i = 0; i < n; i++) {

processes[i].id = i + 1;

cout << "Enter arrival time for process " << i + 1 << ": ";

cin >> processes[i].arrival;

cout << "Enter burst time for process " << i + 1 << ": ";

cin >> processes[i].burst;

}

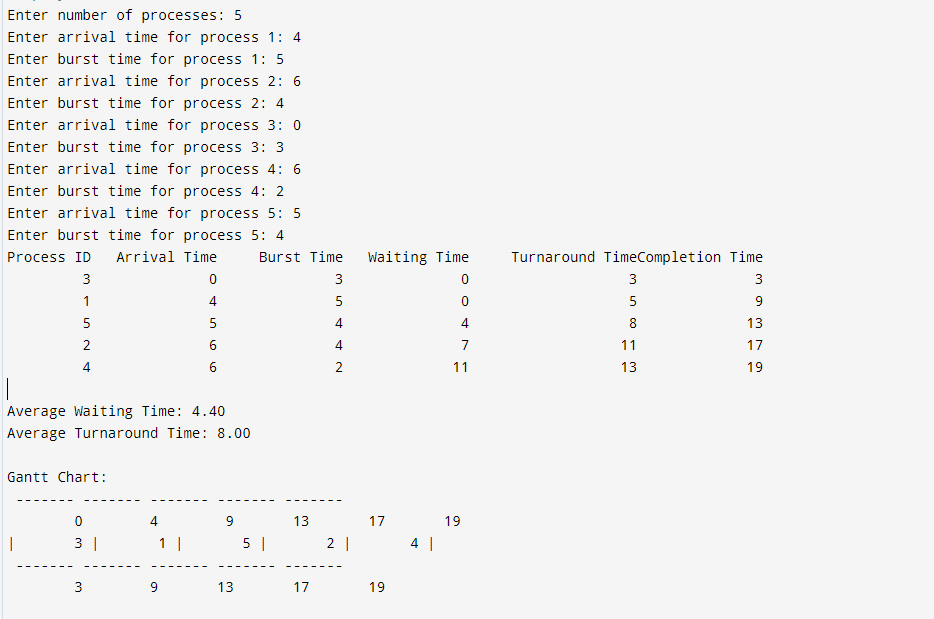
sort(processes.begin(), processes.end(), compareArrival);

findAvgTime(processes);

printGanttChart(processes);

return 0;

}



2) PRIORITY BASED SCHEDULING:

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process {

int id;

int burstTime;

int priority;

int arrivalTime;

int waitingTime;

int turnaroundTime;

};

bool compareByPriority(Process a, Process b) {

if (a.priority == b.priority)

return a.arrivalTime < b.arrivalTime;

return a.priority < b.priority;

}

void calculateTimes(vector<Process>& processes) {

int n = processes.size();

int currentTime = 0;

for (int i = 0; i < n; ++i) {

processes[i].waitingTime = currentTime - processes[i].arrivalTime;

if (processes[i].waitingTime < 0)

processes[i].waitingTime = 0;

processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

currentTime += processes[i].burstTime;

}

}

void printGanttChart(const vector<Process>& processes) {

int n = processes.size();

cout << "\nGantt Chart:\n";

for (int i = 0; i < n; ++i) {

cout << "| P" << processes[i].id << " ";

}

cout << "|\n";

for (int i = 0; i < n; ++i) {

cout << processes[i].waitingTime << "\t";

}

cout << processes[n - 1].turnaroundTime + processes[n - 1].arrivalTime << endl;

}

void calculateAverageTimes(const vector<Process>& processes) {

int n = processes.size();

int totalWaitingTime = 0, totalTurnaroundTime = 0;

for (const auto& process : processes) {

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

}

cout << fixed << setprecision(2);

cout << "\nAverage Waiting Time: " << static\_cast<float>(totalWaitingTime) / n << " ms" << endl;

cout << "Average Turnaround Time: " << static\_cast<float>(totalTurnaroundTime) / n << " ms" << endl;

}

void printProcessTable(const vector<Process>& processes) {

int n = processes.size();

cout << "\nProcess Details:\n";

cout << "-------------------------------------------------------------\n";

cout << "| " << setw(8) << "Process"

<< " | " << setw(12) << "Arrival Time"

<< " | " << setw(10) << "Burst Time"

<< " | " << setw(8) << "Priority"

<< " | " << setw(15) << "Waiting Time"

<< " | " << setw(18) << "Turnaround Time"

<< " |\n";

cout << "-------------------------------------------------------------\n";

for (const auto& process : processes) {

cout << "| " << setw(8) << "P" + to\_string(process.id)

<< " | " << setw(12) << process.arrivalTime

<< " | " << setw(10) << process.burstTime

<< " | " << setw(8) << process.priority

<< " | " << setw(15) << process.waitingTime

<< " | " << setw(18) << process.turnaroundTime

<< " |\n";

}

cout << "-------------------------------------------------------------\n";

}

int main() {

int n;

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

cin >> n;

vector<Process> processes(n);

for (int i = 0; i < n; ++i) {

processes[i].id = i + 1;

cout << "Enter burst time for process P" << i + 1 << ": ";

cin >> processes[i].burstTime;

cout << "Enter priority for process P" << i + 1 << " (smaller number = higher priority): ";

cin >> processes[i].priority;

cout << "Enter arrival time for process P" << i + 1 << ": ";

cin >> processes[i].arrivalTime;

}

sort(processes.begin(), processes.end(), compareByPriority);

calculateTimes(processes);

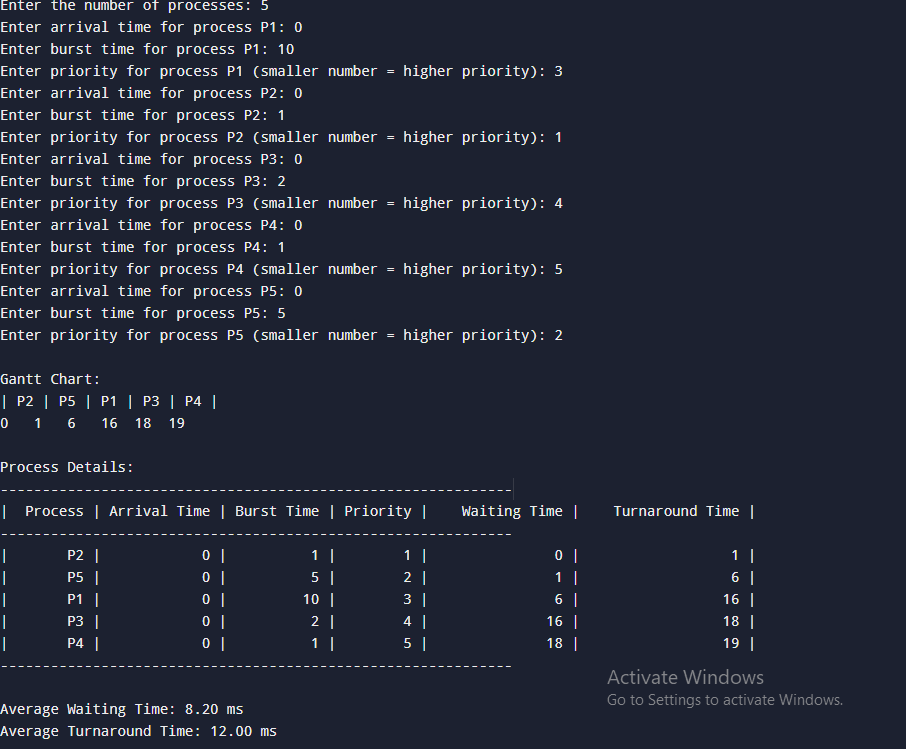
printGanttChart(processes);

printProcessTable(processes);

calculateAverageTimes(processes);

return 0;

}



**Conclusion:**Hence we learned about the non preemptive scheduling algorithms.

**Post Lab Questions**

1. What is a criterion to evaluate a scheduling algorithm?  
     
   A criterion to evaluate a scheduling algorithm includes:

* Throughput: The number of processes completed per unit time. Higher throughput indicates better performance.
* Turnaround Time: The total time taken from the arrival of a process to its completion. Lower turnaround time is generally desirable.
* Waiting Time: The total time a process spends waiting in the ready queue before it gets CPU time. Minimizing waiting time is crucial for efficiency.
* Response Time: The time from the submission of a process to the first response. Shorter response times improve user experience.
* Fairness: Ensuring that all processes get a fair share of CPU time, preventing any single process from monopolizing the CPU.
* CPU Utilization: The percentage of time the CPU is actively working on processes. Higher CPU utilization implies better use of resources.

1. Analyse the efficiency and suitability of FCFS, SJF, and Priority scheduling algorithms.

**First-Come, First-Served (FCFS):**

* Efficiency: Simple to implement but can lead to inefficient CPU utilization and long waiting times, especially if a long process arrives before many short processes (convoy effect).
* Suitability: Best suited for scenarios where fairness is more important than performance, such as batch processing where tasks have similar lengths.

**Shortest Job First (SJF):**

* Efficiency: Generally provides the best average waiting time and turnaround time if job lengths are known in advance. It is optimal in minimizing average waiting time for a given set of processes.
* Suitability: Best suited for environments where process lengths are predictable and where minimizing average waiting time is crucial. It can be problematic with unknown job lengths and may require preemption (Preemptive SJF or Shortest Remaining Time First) to be effective.

**Priority Scheduling:**

* Efficiency: Can be efficient if priorities are well-assigned and processes with higher priorities are handled promptly. However, it can suffer from issues if priorities are not correctly assigned.
* Suitability: Useful when some processes are more important than others. It can be tailored for different use cases based on priority levels.It can lead to starvation if low-priority processes are continuously preempted by higher-priority ones.

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

Starvation:

In the context of Shortest Job First (SJF) scheduling, starvation occurs when longer processes (or jobs) are continually postponed because shorter jobs keep arriving. This can result in the longer processes never getting CPU time, especially in systems where jobs keep arriving and are shorter than the remaining jobs.

Avoidance Strategies:

* 1. Aging: Implementing aging, where the priority of a process increases as it waits longer in the queue, can help prevent starvation. This makes long-waiting processes eventually get a higher priority and be scheduled.
  2. Preemptive Scheduling: Using a preemptive version of SJF, such as Shortest Remaining Time First (SRTF), can help in balancing the scheduling, though it may introduce additional complexity.
  3. Hybrid Scheduling Algorithms: Combining SJF with other scheduling policies, such as a fixed time slice for each process or incorporating priorities, can mitigate the risk of starvation.

**Date: 29/8/24 Signature of faculty in-charge**