**Aim:** To create C/C++ programs for the different scheduling algorithms.

**1. First Come First Serve (FCFS)::**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int waitingTime;

int turnaroundTime;

};

void calculateTimes(vector<Process>& processes) {

processes[0].waitingTime = 0;

processes[0].turnaroundTime = processes[0].burstTime;

int currentTime = processes[0].arrivalTime + processes[0].burstTime;

for (size\_t i = 1; i < processes.size(); ++i) {

processes[i].waitingTime = max(currentTime - processes[i].arrivalTime, 0);

currentTime += processes[i].burstTime;

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

}

}

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 and burst time for P" << processes[i].id << ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime;

}

sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {

return a.arrivalTime < b.arrivalTime;

});

calculateTimes(processes);

cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";

for (const auto& p : processes) {

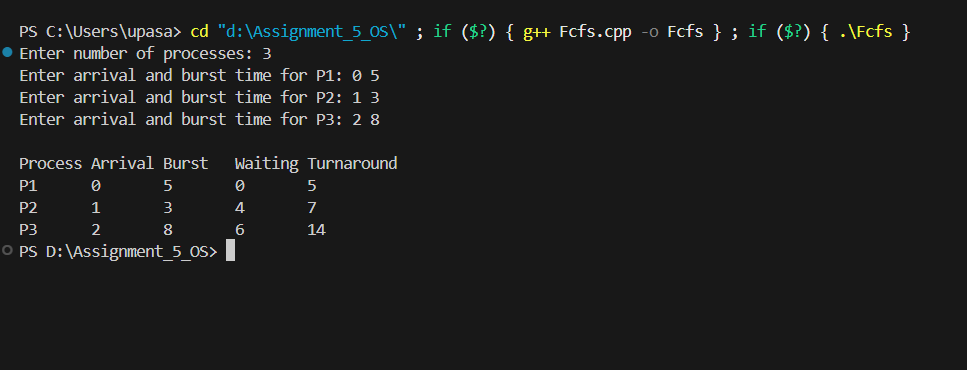
cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime

<< "\t" << p.waitingTime << "\t" << p.turnaroundTime << endl;

}

return 0;

}



**2. Shortest Job First (SJF)::**

#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int completionTime;

int waitingTime;

int turnaroundTime;

};

void sjfPreemptive(vector<Process>& processes) {

int currentTime = 0;

int completed = 0;

int n = processes.size();

while (completed != n) {

int shortest = -1;

int minRemaining = INT\_MAX;

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

if (processes[i].arrivalTime <= currentTime && processes[i].remainingTime < minRemaining && processes[i].remainingTime > 0) {

shortest = i;

minRemaining = processes[i].remainingTime;

}

}

if (shortest == -1) {

currentTime++;

continue;

}

processes[shortest].remainingTime--;

currentTime++;

if (processes[shortest].remainingTime == 0) {

processes[shortest].completionTime = currentTime;

processes[shortest].turnaroundTime = processes[shortest].completionTime - processes[shortest].arrivalTime;

processes[shortest].waitingTime = processes[shortest].turnaroundTime - processes[shortest].burstTime;

completed++;

}

}

}

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 and burst time for P" << processes[i].id << ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime;

processes[i].remainingTime = processes[i].burstTime;

}

sjfPreemptive(processes);

cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";

for (const auto& p : processes) {

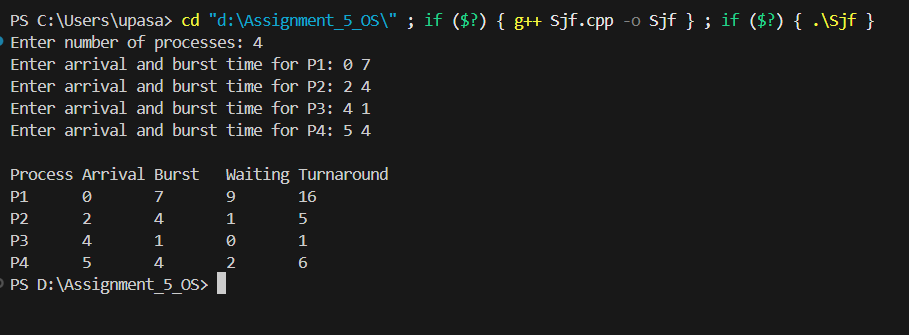
cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime

<< "\t" << p.waitingTime << "\t" << p.turnaroundTime << endl;

}

return 0;

}



**3. Round Robin Scheduling::**

#include <iostream>

#include <vector>

using namespace std;

struct Process {

int id;

int burst\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

};

int main() {

int n, quantum;

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

cin >> n;

cout << "Enter the time quantum: ";

cin >> quantum;

vector<Process> processes(n);

cout << "Enter process ID and burst time for each process (e.g., 0 5):" << endl;

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

int pid, bt;

cin >> pid >> bt;

processes[i].id = pid;

processes[i].burst\_time = bt;

processes[i].remaining\_time = bt;

processes[i].waiting\_time = 0;

}

int time = 0;

bool done;

do {

done = true;

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

if (processes[i].remaining\_time > 0) {

done = false;

if (processes[i].remaining\_time > quantum) {

time += quantum;

processes[i].remaining\_time -= quantum;

} else {

time += processes[i].remaining\_time;

processes[i].waiting\_time = time - processes[i].burst\_time;

processes[i].remaining\_time = 0;

}

}

}

} while (!done);

cout << "\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n";

for (auto& p : processes) {

p.turnaround\_time = p.burst\_time + p.waiting\_time;

cout << "P" << p.id << "\t"

<< p.burst\_time << "\t\t"

<< p.waiting\_time << "\t\t"

<< p.turnaround\_time << "\n";

}

return 0;

}

