**Name : Tanishq Thuse**

**Branch : CS(AI)**

**Year : SY**

**Div : B**

**Roll no. : 60**

**PRN : 12310237**

**Title : OS Assignment-4 Implementation of CPU Scheduling Algorithm**

**Q1) FCFS**

**Code :**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process {

int id; // Process ID

int arrivalTime; // Arrival Time

int burstTime; // Burst Time

int completionTime; // Completion Time

int turnaroundTime; // Turnaround Time

int waitingTime; // Waiting Time

};

bool compareArrival(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}

void displayGanttChart(vector<Process> &processes) {

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

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "|";

for (auto &p : processes) {

cout << " P" << p.id << setw(5) << "|";

}

cout << "\n ";

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "0";

for (auto &p : processes) {

cout << setw(8) << p.completionTime;

}

cout << "\n";

}

int main() {

int n;

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

cin >> n;

float avgWT=0;

float avgTAT=0;

vector<Process> processes(n);

// Input process details

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

processes[i].id = i + 1;

cout << "Enter arrival time and burst time for process P" << processes[i].id

<< ": ";

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

}

// Sort processes based on arrival time

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

// Calculate Completion, Turnaround, and Waiting times

int currentTime = 0;

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

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

currentTime = processes[i].arrivalTime; // Idle until the process arrives

}

processes[i].completionTime = currentTime + processes[i].burstTime;

currentTime = processes[i].completionTime;

processes[i].turnaroundTime = processes[i].completionTime -

processes[i].arrivalTime;

processes[i].waitingTime = processes[i].turnaroundTime -

processes[i].burstTime;

}

for(auto &p:processes)

{

avgTAT+=p.turnaroundTime;

avgWT+=p.waitingTime;

}

avgTAT/=n;

avgWT/=n;

// Display Process Information

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

for (auto &p : processes) {

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

<< p.completionTime << "\t\t" << p.turnaroundTime << "\t\t" <<

p.waitingTime << "\n";

}

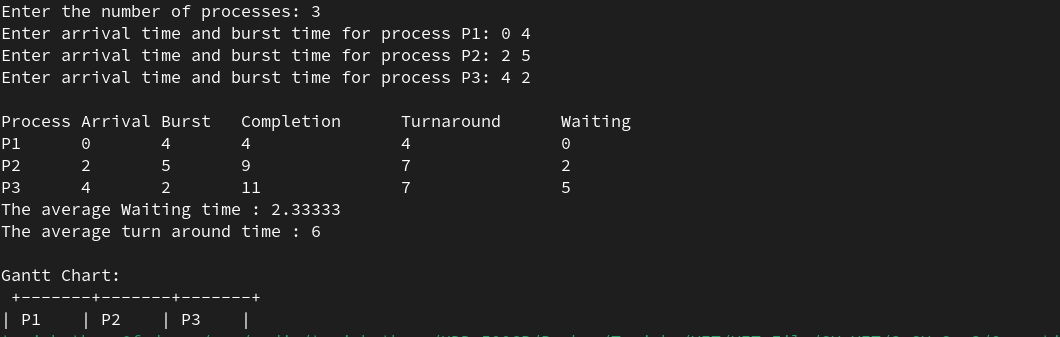
cout<<"The average Waiting time : "<<avgWT<<"\n";

cout<<"The average turn around time : "<<avgTAT<<"\n";

displayGanttChart(processes);

return 0;}

**Output :**



**Q2) SJF**

**Code :**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

#include <limits.h>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

bool isCompleted;

};

bool compareArrival(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}

void displayGanttChart(vector<Process> &processes) {

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

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "|";

for (auto &p : processes) {

cout << " P" << p.id << setw(5) << "|";

}

cout << "\n ";

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "0";

for (auto &p : processes) {

cout << setw(8) << p.completionTime;

}

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 arrival time and burst time for process P" << processes[i].id

<< ": ";

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

processes[i].isCompleted = false;

}

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

int completed = 0, currentTime = 0;

double totalWaitingTime = 0, totalTurnaroundTime = 0;

vector<Process> ganttChart;

while (completed < n) {

int idx = -1;

int minBurstTime = INT\_MAX;

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

if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted) {

if (processes[i].burstTime < minBurstTime) {

minBurstTime = processes[i].burstTime;

idx = i;

}

if (processes[i].burstTime == minBurstTime) {

if (processes[i].arrivalTime < processes[idx].arrivalTime) {

idx = i;

}

}

}

}

if (idx == -1) {

currentTime++;

} else {

processes[idx].completionTime = currentTime + processes[idx].burstTime;

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

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

totalWaitingTime += processes[idx].waitingTime;

totalTurnaroundTime += processes[idx].turnaroundTime;

processes[idx].isCompleted = true;

currentTime = processes[idx].completionTime;

completed++;

ganttChart.push\_back(processes[idx]);

}

}

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

for (auto &p : processes) {

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

<< p.completionTime << "\t\t" << p.turnaroundTime << "\t\t"

<< p.waitingTime << "\n";

}

displayGanttChart(ganttChart);

cout << fixed << setprecision(2);

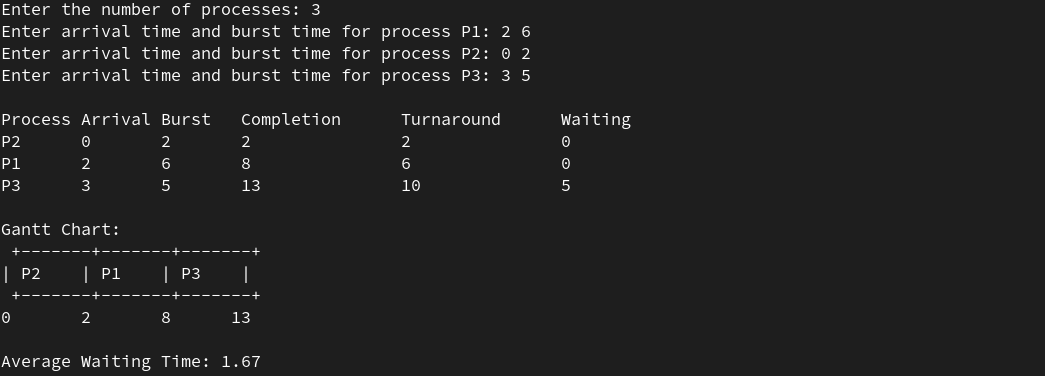
cout << "\nAverage Waiting Time: " << totalWaitingTime / n << endl;

cout << "Average Turnaround Time: " << totalTurnaroundTime / n << endl;

return 0;

}

Output :



**Q3) SRTF**

**Code :**

#include <iostream>

#include <vector>

#include <iomanip>

#include <limits>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int completionTime;

int waitingTime;

int turnaroundTime;

};

void printGanttChart(const vector<int> &ganttChart, int totalTime) {

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

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

cout << "| P" << ganttChart[i] << " ";

}

cout << "|\n0";

for (int i = 1; i <= totalTime; i++) {

cout << " " << i;

}

cout << endl;

}

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

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

cout << setw(5) << "ID" << setw(15) << "Arrival Time"

<< setw(15) << "Burst Time" << setw(20) << "Completion Time"

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

for (const auto &p : processes) {

cout << setw(5) << p.id << setw(15) << p.arrivalTime

<< setw(15) << p.burstTime << setw(20) << p.completionTime

<< setw(15) << p.waitingTime << setw(20) << p.turnaroundTime << endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

void srtf(vector<Process> &processes) {

int n = processes.size();

vector<int> ganttChart;

int completed = 0, currentTime = 0;

double totalWT = 0, totalTAT = 0;

while (completed < n) {

int idx = -1;

int minTime = numeric\_limits<int>::max();

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

if (processes[i].arrivalTime <= currentTime &&

processes[i].remainingTime > 0 &&

processes[i].remainingTime < minTime) {

minTime = processes[i].remainingTime;

idx = i;

}

}

if (idx != -1) {

ganttChart.push\_back(processes[idx].id);

processes[idx].remainingTime--;

currentTime++;

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

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

completed++;

}

} else {

ganttChart.push\_back(0); // 0 represents idle time

currentTime++;

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printGanttChart(ganttChart, currentTime);

printProcessTable(processes, avgWT, avgTAT);

}

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 arrival time and burst time for process P" << i + 1 << ": ";

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

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

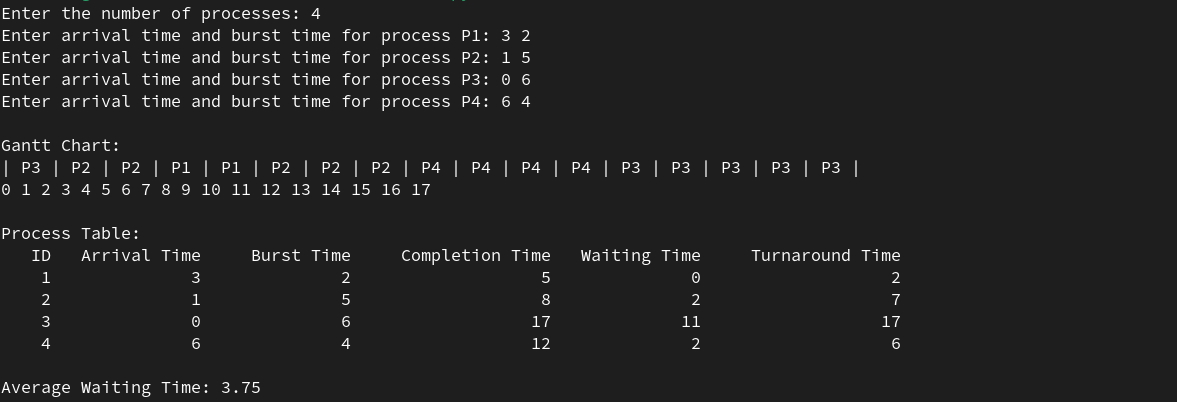
}

srtf(processes);

return 0;

}

**Output :**

****

**Q4) Priority-NonPreemptive :**

**Code :**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int priority;

int completionTime;

int waitingTime;

int turnaroundTime;

};

// Function to print the process table and calculate average times

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

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

cout << setw(5) << "ID"

<< setw(15) << "Arrival Time"

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

<< setw(10) << "Priority"

<< setw(20) << "Completion Time"

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

<< setw(20) << "Turnaround Time"

<< endl;

for (const auto &p : processes) {

cout << setw(5) << p.id

<< setw(15) << p.arrivalTime

<< setw(15) << p.burstTime

<< setw(10) << p.priority

<< setw(20) << p.completionTime

<< setw(15) << p.waitingTime

<< setw(20) << p.turnaroundTime

<< endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

// Priority Non-Preemptive Scheduling Function

void priorityNonPreemptive(vector<Process> &processes) {

int n = processes.size();

vector<int> isCompleted(n, 0);

vector<Process> ganttChart;

double totalWT = 0, totalTAT = 0;

int currentTime = 0, completed = 0;

while (completed < n) {

int idx = -1;

int highestPriority = -1;

// Find the highest priority process that has arrived

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

if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {

if (processes[i].priority > highestPriority) {

highestPriority = processes[i].priority;

idx = i;

}

}

}

if (idx != -1) {

ganttChart.push\_back(processes[idx]);

currentTime += processes[idx].burstTime;

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

isCompleted[idx] = 1;

completed++;

} else {

currentTime++;

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printProcessTable(processes, avgWT, avgTAT);

}

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 arrival time, burst time, and priority for process P" << i + 1 << ": ";

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

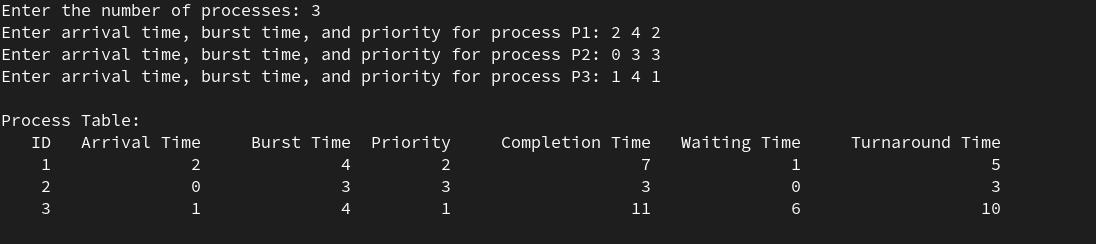
}

priorityNonPreemptive(processes);

return 0;

}

**Output :**



**Q5) Priority-preemptive**

**Code :**

#include <iostream>

#include <vector>

#include <iomanip>

#include <algorithm>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int priority;

int completionTime;

int waitingTime;

int turnaroundTime;

};

// Function to print the process table and calculate average times

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

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

cout << setw(5) << "ID"

<< setw(15) << "Arrival Time"

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

<< setw(10) << "Priority"

<< setw(20) << "Completion Time"

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

<< setw(20) << "Turnaround Time"

<< endl;

for (const auto &p : processes) {

cout << setw(5) << p.id

<< setw(15) << p.arrivalTime

<< setw(15) << p.burstTime

<< setw(10) << p.priority

<< setw(20) << p.completionTime

<< setw(15) << p.waitingTime

<< setw(20) << p.turnaroundTime

<< endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

// Priority Preemptive Scheduling Function

void priorityPreemptive(vector<Process> &processes) {

int n = processes.size();

vector<int> isCompleted(n, 0);

double totalWT = 0, totalTAT = 0;

int currentTime = 0, completed = 0;

int lastExecution = -1;

while (completed < n) {

int idx = -1;

int highestPriority = -1;

// Find the highest priority process that has arrived and is not completed

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

if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {

if (processes[i].priority > highestPriority) {

highestPriority = processes[i].priority;

idx = i;

}

}

}

if (idx != -1) {

if (lastExecution != idx) {

lastExecution = idx;

}

processes[idx].remainingTime--;

currentTime++;

// If the process is completed

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

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

isCompleted[idx] = 1;

completed++;

}

} else {

currentTime++;

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printProcessTable(processes, avgWT, avgTAT);

}

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 arrival time, burst time, and priority for process P" << i + 1 << ": ";

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

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

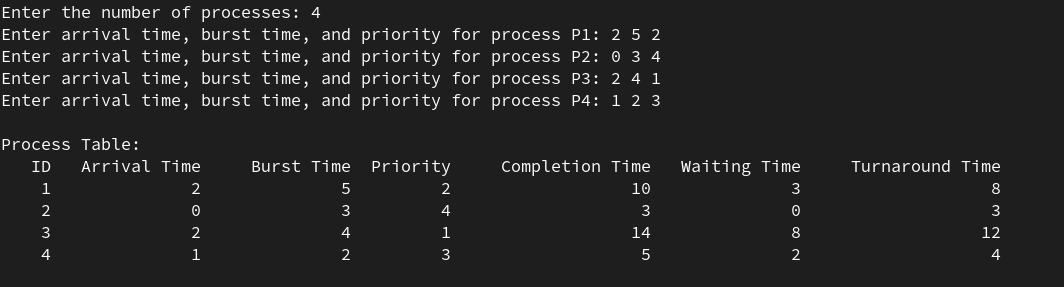
}

priorityPreemptive(processes);

return 0;

}

**Output :**

****

**Q6) Round Robin**

**Code :**

#include <iostream>

#include <vector>

#include <queue>

#include <set>

#include <iomanip>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int completionTime;

int waitingTime;

int turnaroundTime;

};

// Function to print the Gantt Chart

void printGanttChart(const vector<pair<int, int>> &ganttChart) {

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

for (auto &p : ganttChart) {

cout << "| P" << p.first << " ";

}

cout << "|\n0";

for (auto &p : ganttChart) {

cout << " " << p.second;

}

cout << endl;

}

// Function to print the process table and calculate average times

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

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

cout << setw(5) << "ID"

<< setw(15) << "Arrival Time"

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

<< setw(20) << "Completion Time"

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

<< setw(20) << "Turnaround Time"

<< endl;

for (auto &p : processes) {

cout << setw(5) << p.id

<< setw(15) << p.arrivalTime

<< setw(15) << p.burstTime

<< setw(20) << p.completionTime

<< setw(15) << p.waitingTime

<< setw(20) << p.turnaroundTime

<< endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

// Round Robin Scheduling Function

void roundRobin(vector<Process> &processes, int timeQuantum) {

int n = processes.size();

queue<int> q;

set<int> inQueue; // To keep track of processes already in the queue

vector<pair<int, int>> ganttChart;

int currentTime = 0, completed = 0;

double totalWT = 0, totalTAT = 0;

// Add initial processes that have arrived at time 0

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

if (processes[i].arrivalTime <= currentTime) {

q.push(i);

inQueue.insert(i);

}

}

while (completed < n) {

if (q.empty()) {

currentTime++;

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

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

q.push(i);

inQueue.insert(i);

}

}

continue;

}

int idx = q.front();

q.pop();

inQueue.erase(idx);

ganttChart.push\_back({processes[idx].id, currentTime});

// Execute the process for time quantum or remaining time, whichever is smaller

int executionTime = min(timeQuantum, processes[idx].remainingTime);

currentTime += executionTime;

processes[idx].remainingTime -= executionTime;

// Add to the Gantt chart

ganttChart.push\_back({processes[idx].id, currentTime});

// Check if the process is completed

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

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

completed++;

}

// Push the next arrived processes

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

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

q.push(i);

inQueue.insert(i);

}

}

// Reinsert the current process if it is not yet finished

if (processes[idx].remainingTime > 0) {

q.push(idx);

inQueue.insert(idx);

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printGanttChart(ganttChart);

printProcessTable(processes, avgWT, avgTAT);

}

int main() {

int n, timeQuantum;

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 arrival time and burst time for process P" << i + 1 << ": ";

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

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

}

cout << "Enter the time quantum: ";

cin >> timeQuantum;

roundRobin(processes, timeQuantum);

return 0;

}

**Output :**

