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;

}

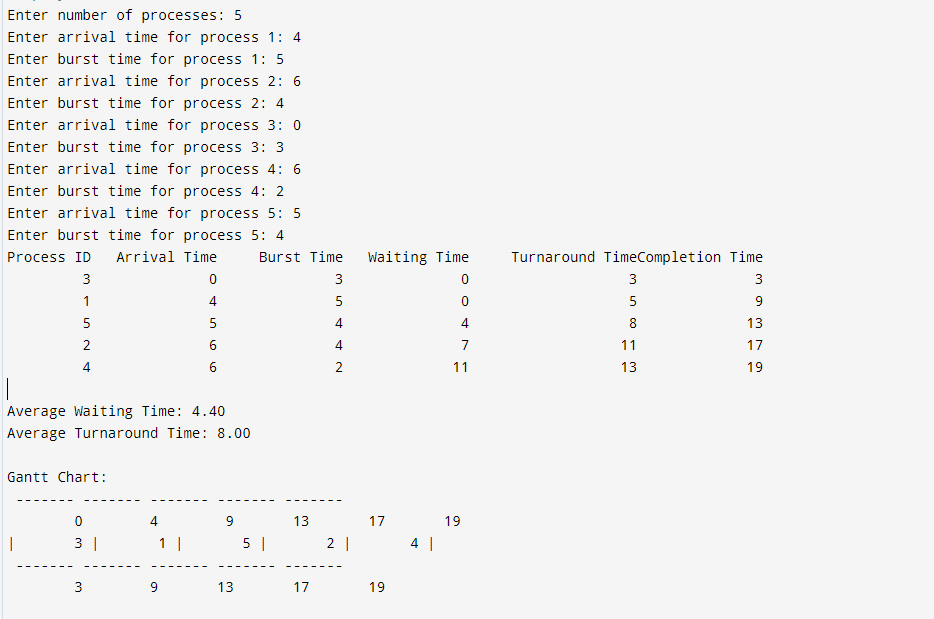
sort(processes.begin(), processes.end(), compareArrival);

findAvgTime(processes);

printGanttChart(processes);

return 0;

}

  
  
  
PRIORITY BASED SCHEDULING ( NON-PREEMPTIVE):

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip> // for std::setw

using namespace std;

struct Process {

int id; // Process ID

int burstTime; // Burst time

int priority; // Priority (smaller number = higher priority)

int arrivalTime; // Arrival time

int waitingTime; // Waiting time

int turnaroundTime; // Turnaround time

};

// Function to sort by priority and arrival time

bool compareByPriority(Process a, Process b) {

if (a.priority == b.priority)

return a.arrivalTime < b.arrivalTime; // If priorities are the same, sort by arrival time

return a.priority < b.priority; // Higher priority first

}

// Function to calculate waiting and turnaround times

void calculateTimes(vector<Process>& processes) {

int n = processes.size();

int currentTime = 0;

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

// Calculate waiting time for the current process

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

if (processes[i].waitingTime < 0)

processes[i].waitingTime = 0;

// Calculate turnaround time for the current process

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

// Update current time

currentTime += processes[i].burstTime;

}

}

// Function to print the Gantt chart

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;

}

// Function to calculate and print average waiting and turnaround times

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;

}

// Function to print the process details table

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

cin >> processes[i].arrivalTime;

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;

}

// Sort processes based on priority and arrival time

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

// Calculate waiting and turnaround times

calculateTimes(processes);

// Print the Gantt chart

printGanttChart(processes);

// Print the process details table

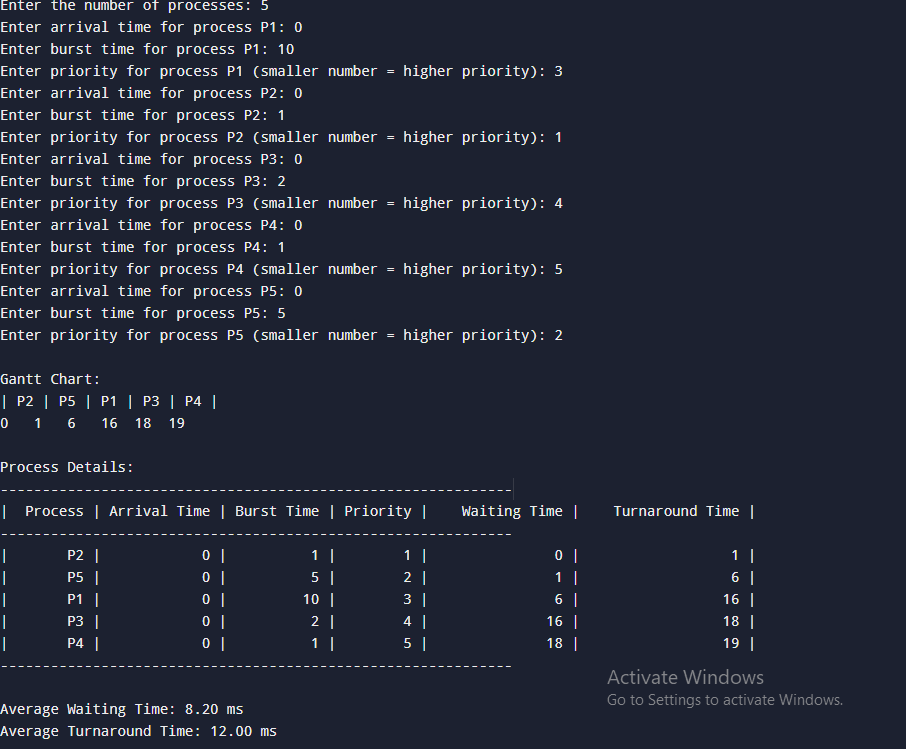
printProcessTable(processes);

// Calculate and print average waiting and turnaround times

calculateAverageTimes(processes);

return 0;

}

  
  
  
  
RoundRobin:

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void roundRobin(int n, vector<int> arrival\_times, vector<int> burst\_times, int time\_quantum) {

vector<int> remaining\_burst\_times = burst\_times; // Copy of burst times

vector<int> completion\_times(n, -1);

vector<int> gantt;

queue<int> q; // To hold the indices of processes

vector<bool> in\_queue(n, false); // To track processes in the queue

int time = 0;

int completed = 0;

int current\_time = 0;

vector<pair<int, int>> gantt\_chart; // To store pairs of (time, process)

while (completed < n) {

// Add all processes that have arrived and are not in the queue

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

if (arrival\_times[i] <= time && !in\_queue[i] && remaining\_burst\_times[i] > 0) {

q.push(i);

in\_queue[i] = true;

}

}

if (q.empty()) {

time++;

continue;

}

int current\_process = q.front();

q.pop();

in\_queue[current\_process] = false;

int execution\_time = min(time\_quantum, remaining\_burst\_times[current\_process]);

remaining\_burst\_times[current\_process] -= execution\_time;

if (gantt\_chart.empty() || gantt\_chart.back().second != current\_process) {

gantt\_chart.push\_back({current\_time, current\_process});

}

current\_time += execution\_time;

// If burst time is finished, record the completion time

if (remaining\_burst\_times[current\_process] == 0) {

completion\_times[current\_process] = current\_time;

completed++;

} else {

q.push(current\_process);

in\_queue[current\_process] = true;

}

}

// Calculate turnaround times and waiting times

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];

}

cout << "Completion Times: ";

for (auto &i : completion\_times) cout << i << " ";

cout << endl;

cout << "Turnaround Times: ";

for (auto &i : turnaround\_times) cout << i << " ";

cout << endl;

cout << "Waiting Times: ";

for (auto &i : waiting\_times) cout << i << " ";

cout << endl;

float avg\_turnaround\_time = 0, avg\_waiting\_time = 0;

for (auto &i : turnaround\_times) avg\_turnaround\_time += i;

for (auto &i : waiting\_times) avg\_waiting\_time += i;

cout << "Avg. Turnaround Time: " << avg\_turnaround\_time / n << endl;

cout << "Avg. Waiting Time: " << avg\_waiting\_time / n << endl;

// Print Gantt Chart

cout << "Gantt Chart\n";

for (auto &entry : gantt\_chart) {

cout << " P" << entry.second + 1;

}

cout << endl;

for (auto &entry : gantt\_chart) {

cout << entry.first << " ";

}

cout << current\_time << endl;

}

int main() {

int n, time\_quantum;

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

cin >> n;

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

cout << "Enter the arrival times of the processes: ";

for (int i = 0; i < n; i++) cin >> arrival\_times[i];

cout << "Enter the burst times of the processes: ";

for (int i = 0; i < n; i++) cin >> burst\_times[i];

cout << "Enter the time quantum: ";

cin >> time\_quantum;

roundRobin(n, arrival\_times, burst\_times, time\_quantum);

return 0;

}

