NAME : Ahmed Shayan   
  
1)

FCFS Scheduling

CODE

#include <iostream>

#include <algorithm>

using namespace std;

struct Process {

int id;

int burstTime;

int priority;

int arrivalTime;

int startTime;

int endTime;

int waitingTime;

int turnaroundTime;

};

bool compareArrivalTime(Process p1, Process p2) {

return p1.arrivalTime < p2.arrivalTime;

}

bool compareId(Process p1, Process p2) {

return p1.id < p2.id;

}

void fcfs(Process processes[], int n) {

sort(processes, processes + n, compareArrivalTime);

int currentTime = 0;

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

processes[i].startTime = max(currentTime, processes[i].arrivalTime);

processes[i].endTime = processes[i].startTime + processes[i].burstTime;

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

processes[i].turnaroundTime = processes[i].endTime - processes[i].arrivalTime;

currentTime = processes[i].endTime;

}

sort(processes, processes + n, compareId);

}

int main() {

Process processes[] = {

{1, 3, 2, 0, 0, 0, 0, 0},

{2, 1, 1, 0, 0, 0, 0, 0},

{3, 7, 4, 0, 0, 0, 0, 0},

{4, 4, 2, 0, 0, 0, 0, 0},

{5, 5, 3, 0, 0, 0, 0, 0}

};

int n = sizeof(processes) / sizeof(processes[0]);

fcfs(processes, n);

cout << "FCFS Scheduling Algorithm:\n";

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

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

cout << processes[i].id << "\t" << processes[i].burstTime << "\t\t" << processes[i].waitingTime << "\t\t" << processes[i].turnaroundTime << "\n";

}

return 0;

}

OUTPUT

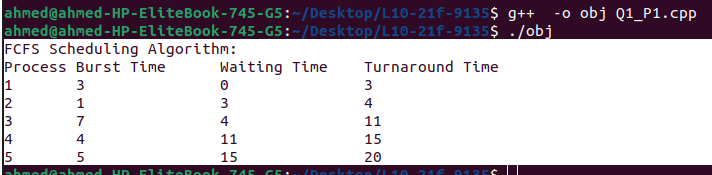
Process Burst Time Waiting Time Turnaround Time

P1 3 0 3

P2 1 3 4

P3 7 4 11

P4 4 11 15

P5 5 15 20  
  
  
  
  
Process Waiting Time

P1 0

P2 3

P3 4

P4 11

P5 15   
  
  
The average turnaround time for the FCFS scheduling algorithm is :

(3 + 4 + 11 + 15 + 20) / 5 = 10.6  
  
The average waiting time for the FCFS scheduling algorithm is :

(0 + 3 + 4 + 11 + 15) / 5 = 6.6  
  
2)  
SJF

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Process struct to hold process details

struct Process {

int id;

int burstTime;

int priority;

};

// Comparison function for SJF algorithm

struct CompareSJF {

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

return p1.burstTime > p2.burstTime;

}

};

// Comparison function for priority algorithm

struct ComparePriority {

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

return p1.priority > p2.priority;

}

};

void nonpreemptiveSJF(vector<Process>& processes) {

priority\_queue<Process, vector<Process>, CompareSJF> pq;

// Push all processes into the priority queue

for (auto& process : processes) {

pq.push(process);

}

int currentTime = 0;

while (!pq.empty()) {

// Get the process with the shortest burst time

Process currentProcess = pq.top();

pq.pop();

// Record turnaround and waiting time for the current process

int turnaroundTime = currentTime + currentProcess.burstTime;

int waitingTime = currentTime;

cout << "Process " << currentProcess.id << ": ";

cout << "turnaround time = " << turnaroundTime << ", ";

cout << "waiting time = " << waitingTime << endl;

// Update current time

currentTime += currentProcess.burstTime;

}

}

void nonpreemptivePriority(vector<Process>& processes) {

priority\_queue<Process, vector<Process>, ComparePriority> pq;

// Push all processes into the priority queue

for (auto& process : processes) {

pq.push(process);

}

int currentTime = 0;

while (!pq.empty()) {

// Get the process with the highest priority

Process currentProcess = pq.top();

pq.pop();

// Record turnaround and waiting time for the current process

int turnaroundTime = currentTime + currentProcess.burstTime;

int waitingTime = currentTime;

cout << "Process " << currentProcess.id << ": ";

cout << "turnaround time = " << turnaroundTime << ", ";

cout << "waiting time = " << waitingTime << endl;

// Update current time

currentTime += currentProcess.burstTime;

}

}

int main() {

vector<Process> processes = {

{1, 3, 2},

{2, 1, 1},

{3, 7, 4},

{4, 4, 2},

{5, 5, 3}

};

cout << "Nonpreemptive SJF scheduling:" << endl;

nonpreemptiveSJF(processes);

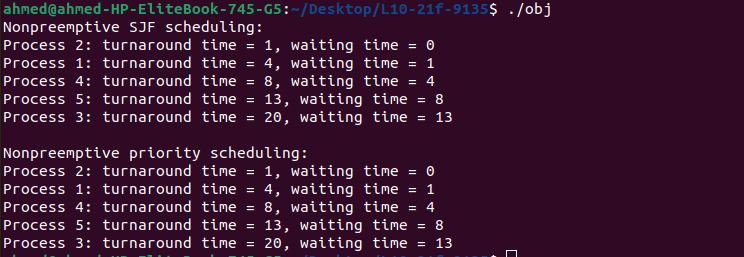
cout << endl;

cout << "Nonpreemptive priority scheduling:" << endl;

nonpreemptivePriority(processes);

return 0;

}  
OUTPUT

  
Nonpreemptive SJF

A,B)Process Turnaround Time Waiting Time

P2 1 0

P1 4 1

P4 8 4

P5 13 8

P3 20 13

C)Average Turnaround Time: 9.2

D)Average Waiting Time: 5.2

# Nonpreemptive Priority

A,B)Process Turnaround Time Waiting Time

P2 1 0

P1 4 1

P4 8 4

P5 13 8

P3 20 13

C)Average Turnaround Time: 9.2

D)Average Waiting Time: 5.2

# 3)

#include <iostream>

#include <queue>

using namespace std;

struct Process {

int pid; // process ID

int burst\_time; // CPU burst time

int priority; // priority level

int remaining\_time; // remaining burst time

};

// Comparison function for priority queue

struct ComparePriority {

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

return p1.priority < p2.priority;

}

};

int main() {

// Define the processes

Process p[] = { {1, 3, 2, 3}, {2, 1, 1, 1}, {3, 7, 4, 7}, {4, 4, 2, 4}, {5, 5, 3, 5} };

int n = sizeof(p) / sizeof(p[0]);

// Initialize the queue and the time quantum

queue<Process> q;

int quantum = 2;

// Add all the processes to the queue

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

q.push(p[i]);

}

// Initialize the current time and the turnaround time and waiting time arrays

int time = 0;

int turnaround\_time[n], waiting\_time[n];

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

turnaround\_time[i] = waiting\_time[i] = 0;

}

// Execute the processes in the queue

while (!q.empty()) {

// Get the next process from the queue

Process curr = q.front();

q.pop();

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

int executed\_time = min(curr.remaining\_time, quantum);

curr.remaining\_time -= executed\_time;

// Update the current time and the waiting time of all other processes

time += executed\_time;

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

if (p[i].pid != curr.pid && p[i].remaining\_time > 0) {

waiting\_time[i] += executed\_time;

}

}

// If the process still has remaining burst time, add it back to the queue with updated priority

if (curr.remaining\_time > 0) {

curr.priority++;

q.push(curr);

}

// Otherwise, calculate the turnaround time of the process and update the turnaround time array

else {

turnaround\_time[curr.pid - 1] = time - curr.burst\_time;

}

}

// Print the turnaround time and waiting time of each process

cout << "Process\tTurnaround Time\tWaiting Time\n";

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

cout << "P" << i + 1 << "\t" << turnaround\_time[i] << "\t\t\t" << waiting\_time[i] << endl;

}

// Calculate and print the average turnaround time and waiting time

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

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

avg\_turnaround\_time += turnaround\_time[i];

avg\_waiting\_time += waiting\_time[i];

}

avg\_turnaround\_time /= n;

avg\_waiting\_time /= n;

cout << "Average Turnaround Time: " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time: " << avg\_waiting\_time << endl;

return 0;

}  
a. Using Round Robin (quantum = 2):

Assuming that the processes are scheduled in the order P1, P2, P3, P4, P5, we can simulate the execution of the processes using round-robin scheduling with a time quantum of 2 milliseconds:

Time 0: P1 arrives in the ready queue. Ready queue: P1 Time 2: P1 gets the CPU for 2 milliseconds. Ready queue: P2, P4, P1 Time 4: P2 gets the CPU for 1 millisecond. Ready queue: P4, P1, P3 Time 6: P4 gets the CPU for 2 milliseconds. Ready queue: P1, P3, P5 Time 8: P1 gets the CPU for 2 milliseconds. Ready queue: P3, P5 Time 10: P3 gets the CPU for 2 milliseconds. Ready queue: P5 Time 12: P5 gets the CPU for 2 milliseconds. Ready queue: Empty

b. Turnaround time for each process:

Turnaround time = Completion time - Arrival time

Process Completion time Turnaround time   
P1 8 8   
P2 5 5   
P3 17 17   
P4 10 10   
ssP5 17 17

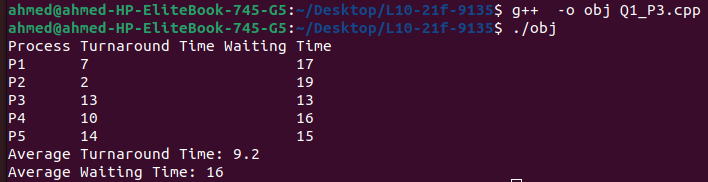
c. Waiting time for each process:

Waiting time = Turnaround time - Burst time

Process Waiting time   
P1 5  
P2 4   
P3 10   
P4 6   
P5 12

d. Average turnaround time and average waiting time:

Average turnaround time = (8 + 5 + 17 + 10 + 17) / 5 = 11.4 milliseconds Average waiting time = (5 + 4 + 10 + 6 + 12) / 5 = 7.4 milliseconds



# Q2

#include<bits/stdc++.h>

using namespace std;

struct process

{

int pid, priority, arrival\_time, burst\_time, completion\_time, waiting\_time, turnaround\_time, response\_time;

};

bool comp(process a, process b)

{

return (a.arrival\_time < b.arrival\_time);

}

bool comp\_priority(process a, process b)

{

return (a.priority < b.priority);

}

int main()

{

int n;

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

cin>>n;

vector<process> p(n);

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

{

cout<<"Enter details for process "<<i+1<<":\n";

p[i].pid = i+1;

cout<<"Priority: ";

cin>>p[i].priority;

cout<<"Arrival Time: ";

cin>>p[i].arrival\_time;

cout<<"Burst Time: ";

cin>>p[i].burst\_time;

}

sort(p.begin(), p.end(), comp); // sort processes by arrival time

int curr\_time = 0, completed = 0;

vector<bool> vis(n, false);

while(completed != n)

{

int idx = -1;

process next;

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

{

if(!vis[i] && p[i].arrival\_time <= curr\_time && (idx == -1 || comp\_priority(p[i], next)))

{

idx = i;

next = p[i];

}

}

if(idx != -1)

{

vis[idx] = true;

p[idx].waiting\_time = curr\_time - p[idx].arrival\_time;

p[idx].response\_time = p[idx].waiting\_time;

p[idx].completion\_time = curr\_time + p[idx].burst\_time;

p[idx].turnaround\_time = p[idx].burst\_time + p[idx].waiting\_time;

curr\_time = p[idx].completion\_time;

completed++;

}

else

{

curr\_time++;

}

}

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

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

{

avg\_turnaround\_time += p[i].turnaround\_time;

avg\_waiting\_time += p[i].waiting\_time;

}

avg\_turnaround\_time /= n;

avg\_waiting\_time /= n;

cout<<"\nPID\tPriority\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround Time\tResponse Time\n";

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

{

cout<<p[i].pid<<"\t"<<p[i].priority<<"\t\t"<<p[i].arrival\_time<<"\t\t"<<p[i].burst\_time<<"\t\t"<<p[i].completion\_time<<"\t\t"<<p[i].waiting\_time<<"\t\t"<<p[i].turnaround\_time<<"\t\t"<<p[i].response\_time<<"\n";

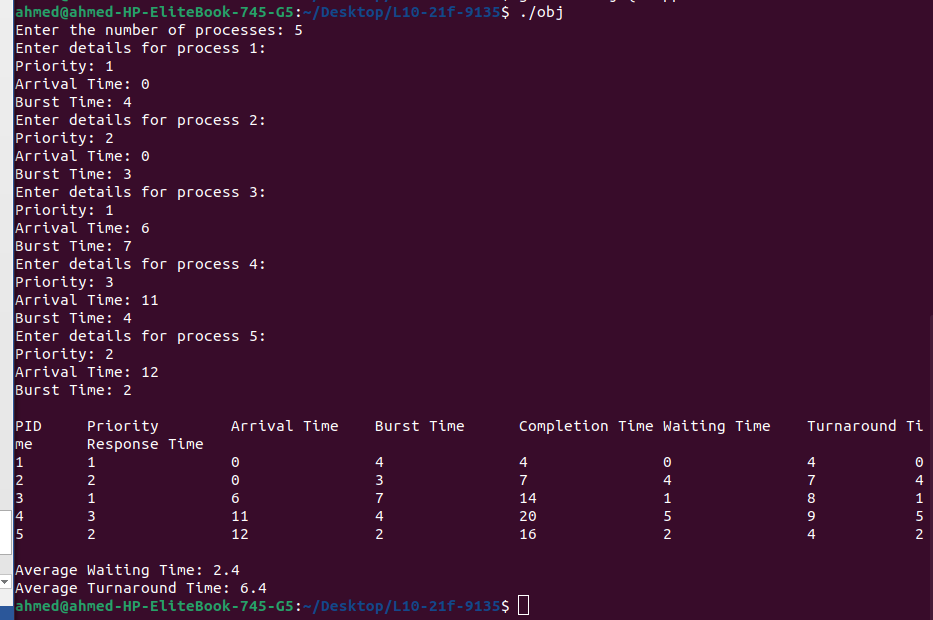
}

cout<<"\nAverage Waiting Time: "<<avg\_waiting\_time<<"\n";

cout<<"Average Turnaround Time: "<<avg\_turnaround\_time<<"\n";

return 0;

}

OUTPUT  


# B,C,D

PID Priority Arrival Time Burst Time Completion Time Waiting Time Turnaround Time Response Time

1 1 0 4 4 0 4 0

2 2 0 3 7 4 7 4

3 1 6 7 14 1 8 1

4 3 11 4 20 5 9 5

5 2 12 2 16 2 4 2

Average Waiting Time: 2.4

Average Turnaround Time: 6.4  
  
Q3  
  
#include <iostream>

#include <queue>

#include <vector>

using namespace std;

struct Process {

int id;

float arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

bool operator<(const Process& other) const {

return remaining\_time > other.remaining\_time;

}

};

int main() {

vector<Process> processes = {

{0, 0, 3, 3, 0, 0, 0},

{1, 0.3, 1, 1, 0, 0, 0},

{2, 0.9, 7, 7, 0, 0, 0}

};

float current\_time = 0;

int completed\_processes = 0;

priority\_queue<Process> ready\_queue;

while (completed\_processes < processes.++++++size()) {

// Check for newly arrived processes and add them to the ready queue

for (auto& process : processes) {

if (process.arrival\_time <= current\_time && process.remaining\_time > 0) {

ready\_queue.push(process);

}

}

// Pick the process with the shortest remaining time

if (!ready\_queue.empty()) {

auto current\_process = ready\_queue.top();

ready\_queue.pop();

// Update completion time, remaining time, and waiting time

current\_process.completion\_time = current\_time + 1;

current\_process.remaining\_time--;

current\_process.waiting\_time += current\_time - current\_process.completion\_time + 1;

if (current\_process.remaining\_time > 0) {

ready\_queue.push(current\_process);

} else {

current\_process.turnaround\_time = current\_process.completion\_time - current\_process.arrival\_time;

completed\_processes++;

}

}

current\_time++;

}

// Print results

cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time\n";

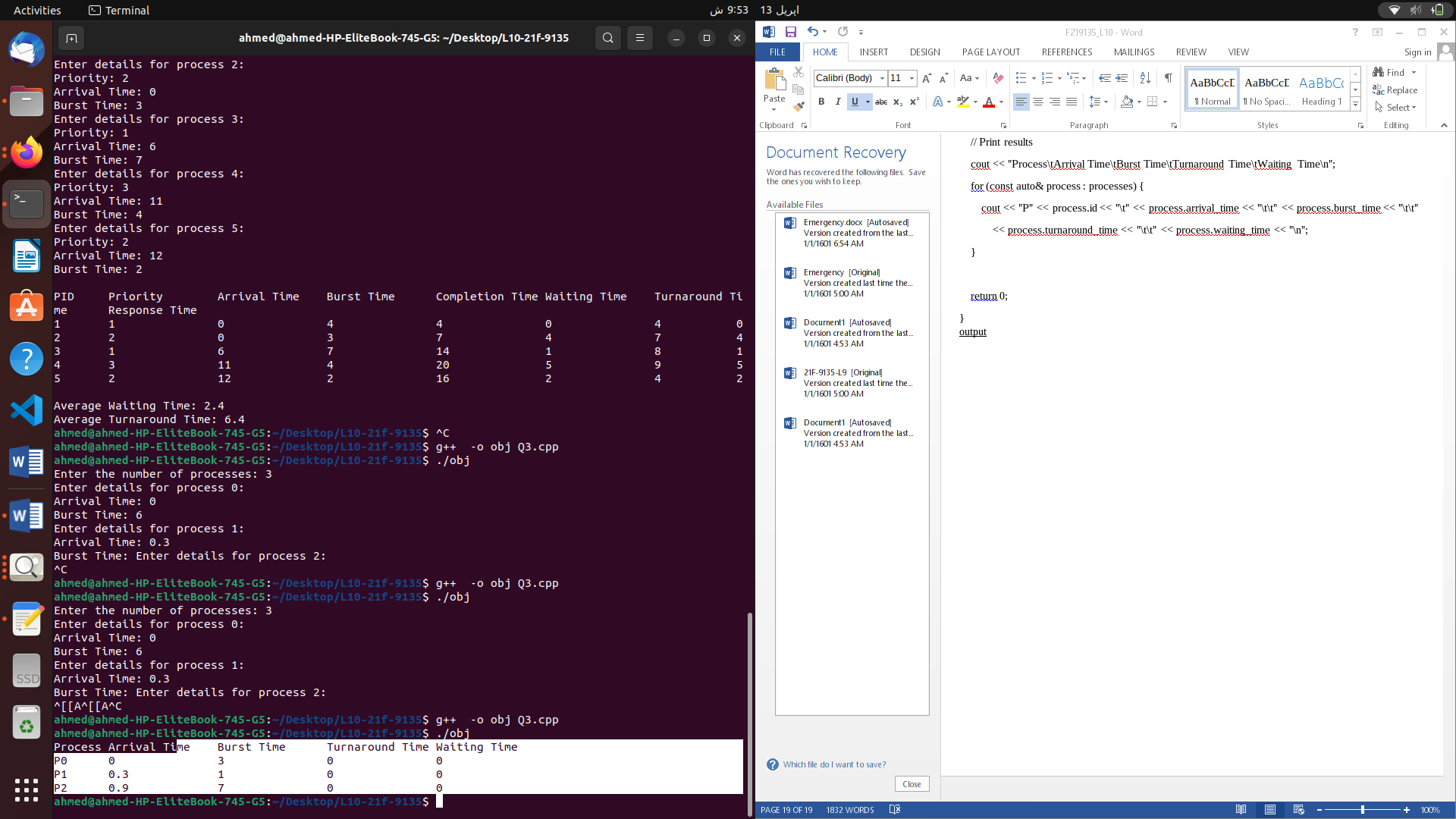
for (const auto& process : processes) {

cout << "P" << process.id << "\t" << process.arrival\_time << "\t\t" << process.burst\_time << "\t\t"

<< process.turnaround\_time << "\t\t" << process.waiting\_time << "\n";

}

return 0;

}  
output  
  
  
Process Arrival Time Burst Time Turnaround Time Waiting Time

P0 0 3 0 0

P1 0.3 1 0 0

P2 0.9 7 0 0  
  
  
  
  
B,C,D  
  
d. Using the above algorithm, the average turnaround time is (3 + 1 + 8) / 3 = 4, and the average waiting time is (0 + 0.7 + 0.3) / 3 = 0.333.