**Reg No.: EG/2020/3894**

**Name : Didulani P.K.S**

**Date : 01/06/2024**

**Implement the FCFS scheduling algorithm.**

#include<iostream>

using namespace std;

#include<vector>

#include<map>

int main(){

    // get the number of processes

    int number\_of\_processes;

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

    cin >> number\_of\_processes;

    cout <<endl;

    // map to store the arrival time and burst time of each process. Here key is arrival time and value is burst time

    map<int, int> process;

    // Get the arrival time and burst time of each process in a single line

    cout << "Enter the arrival time and burst time of process separated by space: "<<endl;

    for (int i = 0; i < number\_of\_processes; i++) {

        int arrival\_time, burst\_time;

        cin >> arrival\_time >> burst\_time;

        process[arrival\_time] = burst\_time;

    }

// array to store the completion time of each process

    vector<int> completion\_time(number\_of\_processes);

    // get the completion time of each process

    int temp=0;

    int index =0;

    for(const auto& i: process){

        if(i.first == 0){

            temp=  i.second;

        }else{

            temp = i.second + temp;

        }

        completion\_time[index] = temp;

        index++;

    }

    // array to store the turnaround time of each process

    vector<int> turnaround\_time(number\_of\_processes);

    // get the turnaround time of each process

    int temp1=0;

    for (auto i: process){

        turnaround\_time[temp1] = completion\_time[temp1] - i.first;

        temp1++;

    }

    // array to store the waiting time of each process

    vector<int> waiting\_time(number\_of\_processes);

    // get the waiting time of each process

    int temp2=0;

    for (auto i: process){

        waiting\_time[temp2] = turnaround\_time[temp2] - i.second;

        temp2++;

    }

    // print the average waiting times

    int sum =0;

    for(int i=0; i<number\_of\_processes; i++){

        sum = sum + waiting\_time[i];

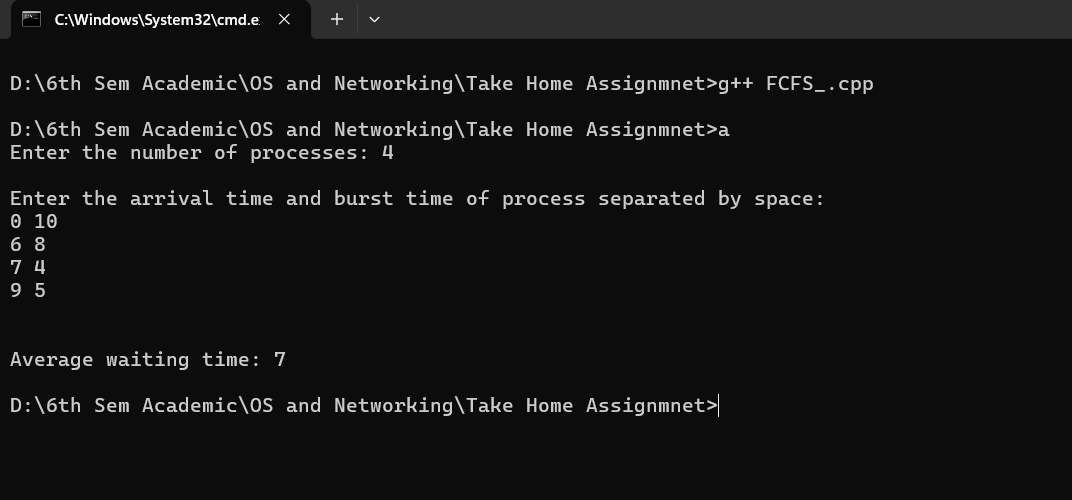
    }

    double average\_waiting\_time = sum/number\_of\_processes;

    cout << "\n\nAverage waiting time: " << average\_waiting\_time <<endl;

}

**Output of the FCFS algorithm,**



**Implement the SJF scheduling algorithm.**

#include <iostream>

#include <algorithm>

#include <cstring>

using namespace std;

int main() {

    // Define a structure to represent each process

    typedef struct process {

        int arrival\_time, burst\_time, completion\_time, turnaround\_time, waiting\_time, btt;

        string pro\_id; // Process ID

    } Schedule;

    // Comparison function to sort processes by arrival time

    auto compare = [](Schedule a, Schedule b) {

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

    };

    // Comparison function to sort processes by burst time

    auto compare2 = [](Schedule a, Schedule b) {

        return a.burst\_time < b.burst\_time;

    };

    Schedule pro[10]; // Array to store processes

    int n, i, j, pcom; // n: number of processes, pcom: processes completed

//i & j = iterative variables

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

    cin >> n;

    cout << "Enter the arrival time and burst time of process separated by space: \n";

    // Input process details

    for (i = 0; i < n; i++) {

        cin >> pro[i].arrival\_time;

        cin >> pro[i].burst\_time;

        pro[i].btt = pro[i].burst\_time;

    }

 // Sort processes by their arrival time

    sort(pro, pro + n, compare);

    i = 0;

    pcom = 0;

    // Loop until all processes are completed

    while (pcom < n) {

        // Find the processes that have arrived by the current time

        for (j = 0; j < n; j++) {

            if (pro[j].arrival\_time > i)

                break;

        }

        // Sort the arrived processes by burst time

        sort(pro, pro + j, compare2);

        // If there are processes that have arrived

        if (j > 0) {

            // Find the first process that is not yet completed

            for (j = 0; j < n; j++) {

                if (pro[j].burst\_time != 0)

                    break;

            }

            // If the next process's arrival time is in the future, skip time to its arrival

            if (pro[j].arrival\_time > i) {

                i = pro[j].arrival\_time;

            }

            // Update the completion time of the current process

            pro[j].completion\_time = i + 1;

            // Decrement the remaining burst time of the current process

            pro[j].burst\_time--;

        }

        i++; // Increment current time

        pcom = 0; // Reset completed process count

        // Count the number of processes that are completed

        for (j = 0; j < n; j++) {

            if (pro[j].burst\_time == 0)

                pcom++;

        }

    }

double sum = 0;

    double avg\_wait\_time;

    // Calculate turnaround time and waiting time for each process

    for (i = 0; i < n; i++) {

        pro[i].turnaround\_time = pro[i].completion\_time - pro[i].arrival\_time;

        pro[i].waiting\_time = pro[i].turnaround\_time - pro[i].btt;

        sum += pro[i].waiting\_time; // Add waiting time to sum

    }

    // Calculate average waiting time

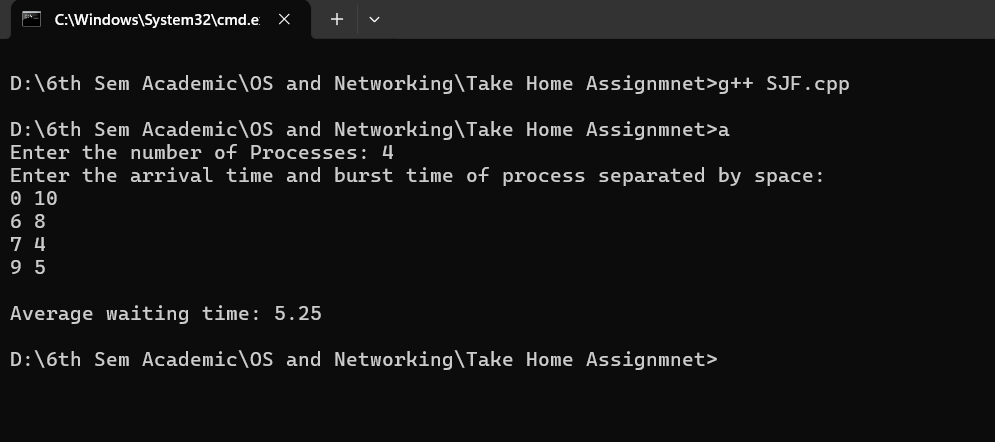
    avg\_wait\_time = sum / n;

    cout << "\nAverage waiting time: " << avg\_wait\_time << endl;

    return 0;

}

**Output of the SJF algorithm**

****

**Implement the RR scheduling algorithm.**

#include <iostream>

#include <vector>

using namespace std;

int main() {

    int i, n, time, remain, temps = 0, time\_quantum;

    // wt=Total waiting time, tat=Total turnaround time

    int wt = 0, tat = 0;

    // get number of processes

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

    cin >> n;

    // Initialize the remaining processes counter

    remain = n;

    // Define vector arrays for arrival time, burst time, and remaining time for each process

    vector<int> at(n);

    vector<int> bt(n);

    vector<int> rt(n);

    // get the arrival time and burst time of each process

    cout << "Enter the arrival time and burst time of processes separated by space:" << endl;

    for (i = 0; i < n; i++) {

        cin >> at[i] >> bt[i];

        rt[i] = bt[i]; // Initialize remaining time with burst time

    }

    // Get the time quantum

    cout << "Enter the value of time QUANTUM: ";

    cin >> time\_quantum;

 // Start the scheduling algorithm

    for (time = 0, i = 0; remain != 0;) {

        // If remaining time for the current process is less than or equal to time quantum and greater than 0

        if (rt[i] <= time\_quantum && rt[i] > 0) {

            // Update time with the remaining time for the current process

            time += rt[i];

            // Set remaining time for the current process to 0

            rt[i] = 0;

            // Set flag to indicate that the process has finished executing

            temps = 1;

        } else if (rt[i] > 0) {

            // If remaining time for the current process is greater than 0, decrement remaining time by time quantum

            rt[i] -= time\_quantum;

            // Update time with the time quantum

            time += time\_quantum;

        }

        // If the current process has finished executing

        if (rt[i] == 0 && temps == 1) {

            // Decrement the remaining processes counter

            remain--;

            // turnaround time for the current process

            int turnaround\_time = time - at[i];

            // waiting time for the current process

            int waiting\_time = turnaround\_time - bt[i];

            // Update total waiting time

            wt += waiting\_time;

            // Reset flag indicating that the process has finished executing

            temps = 0;

        }

        // Determine the next process to execute based on arrival time and time quantum

        if (i == n - 1) {

            i = 0; // Wrap around to the beginning of the process queue if reached the end

        }

        // check if the arrival time is less than or equal to the current time

        else if (at[i + 1] <= time) {

            i++; // Move to the next process if so

        }

        else {

            i = 0; // otherwise reset to the beginning of the process queue

        }

    }

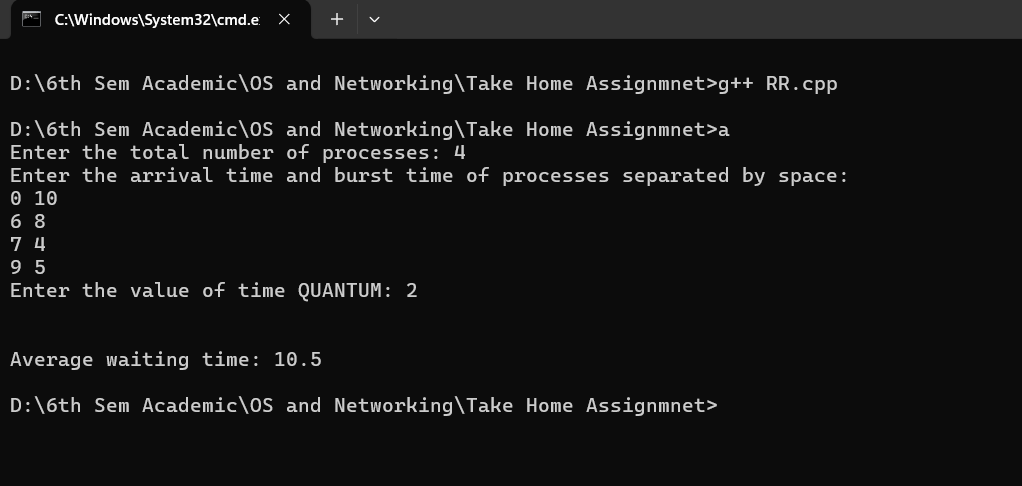
// Output average waiting time

    cout << "\n\nAverage waiting time: " << (double)wt / n << endl;

    return 0;

}

**Output of the RR algorithm,**

****

**Analysis comparing the performance of the algorithms based on average waiting time.**

Given Data,

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | Burst Time |
| P1 | 0 | 10 |
| P2 | 6 | 8 |
| P3 | 7 | 4 |
| P4 | 9 | 5 |

Form Results,

|  |  |
| --- | --- |
| CPU Scheduling Algorithm | Average Waiting Time (units) |
| First Come First Server (FCFS) | 7 |
| Shortest Job First | 5.25 |
| Round Rodin (RR) for 2 units time quantum | 10.5 |

Here only the FCFS scheduling algorithm is non-preemptive, Other 2 algorithms are preemptive.

From the Results we can see the Shortest Job First has the smallest Average waiting time.

* In the SJF the process which have the shortest burst time is execute first. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. So, we can say SJS prioritizes shorter jobs first. Therefore, it leads to quicker turnaround times for processes overall.

Turn Around Time = Completion Time – Arrival Time

* Since shorter jobs are executed first, they typically complete faster. So, it leads the lower turnaround time for individual processes.

Waiting Time = Turn Around Time – Burst Time

* When the turn around time getting smaller waiting time also getting smaller for each individual processes. Finally, the Overall average waiting time also become small. That is the advantage of the SJF. From this it minimizes the wasted CPU time because it always selected job available, leading to efficient CPU utilization.
* In the FCFS algorithm CPU execute the processes which come first to the CPU. It is not prioritized shorter jobs, it leads longer waiting times. Here in the 7th unit of time the remaining burst time of the P2 is 7 but the P3 has 4 units of burst time. But in the FCFS the P3 have to wait until the P2 terminate. Therefore, it will lead the higher waiting time (Convoy Effect).
* Here the Round Rodin algorithm has the highest waiting time by compare to the other 2 algorithms. This is because processes are given equal time slices regardless of their burst times, potentially leading to longer waiting times for shorter processes. Also, here we use the time quantum is 2 units. Which is small, that also may impact to the overall system performance because here some processes burst times are large. From the Gantt chart above it prove.

**Gantt Chart for FCFS algorithm,**

0 10 18 22 27

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P3 | P4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Complete Time | Turnaround Time | Waiting Time |
| P1 | 0 | 10 | 10 | 10 | 0 |
| P2 | 6 | 8 | 18 | 12 | 4 |
| P3 | 7 | 4 | 22 | 15 | 11 |
| P4 | 9 | 5 | 27 | 18 | 13 |

**Gantt Chart for SJF algorithm,**

0 10 14 19 27

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P3 | P4 | P2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Total waiting Time | Process executed before | Waiting Time |
| P1 | 0 | 10 | 0 | 0 | 0 |
| P2 | 6 | 8 | 19 | 0 | 13 |
| P3 | 7 | 4 | 10 | 0 | 3 |
| P4 | 9 | 5 | 14 | 0 | 5 |

**Gantt Chart for RR algorithm,**

0 6 8 10 12 14 16 18 20 22 24 26 27

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P1 | P3 | P2 | P4 | P1 | P3 | P2 | P4 | P2 | P4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | Complete Time | Turnaround Time | Waiting Time |
| P1 | 0 | 10 | 18 | 18 | 8 |
| P2 | 6 | 8 | 26 | 20 | 12 |
| P3 | 7 | 4 | 20 | 13 | 9 |
| P4 | 9 | 5 | 17 | 18 | 13 |