LAB REPORT

COURSE TITLE	:	Operating Systems Lab
COURSE CODE	:	CSE 210
LAB REPORT NO.	:	07
SUBMISSION DATE	:	07-10-2025

SUBMITTED TO

NAME	:	Mishal Al Rahman
DEPT. OF	:	Computer Science and Engineering (CSE) Bangladesh University of Business & Technology (BUBT)

SUBMITTED BY

NAME	:	Shadman Shahriar
ID NO.	:	20245103408
INTAKE	:	53
SECTION	:	1
PROGRAM	:	B.Sc. Engg. in CSE

Scheduling Algorithms

Objective: Writing codes in Java or in **C/C++** to simulate various Process Scheduling Algorithms.

Since we have taken courses on **C** and **C++** in our previous semesters, I will write the scheduling algorithms in **C++**.

First Come First Serve (FCFS)

The First Come First Serve (**FCFS**) is a **non-preemptive** scheduling algorithm where tasks are processed in the exact order of their arrival, similar to people in a queue. Due to its non-preemptive nature, it is the simplest scheduling algorithm that schedules processes in the order they arrive, without considering priority or other factors.

Here is my implementation of the **FCFS** algorithm in **C++**:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct process
{
    int process_id = 0;
    int burst_time = 0;
    int arrival_time = 0;
    int completion_time = 0;
    int waiting_time = 0;
    int turn_around_time = 0;
};
```

```
bool sortByArrival(const process &p1, const process &p2)
{
    return p1.arrival_time < p2.arrival_time;
}

bool sortByID(const process &p1, const process &p2)
{
    return p1.process_id < p2.process_id;
}

int main()
{
    int n;
    vector<process> processes = {};

    cout << "Enter the number of processes: ";
    cin >> n;
    for (int i = 0; i < n; i++)
        processes.push_back({i + 1});

    cout << endl;
    cout << "Enter the CPU times" << endl;
    for (int i = 0; i < n; i++)
        cin >> processes[i].burst_time;

    cout << endl;
    cout << "Enter the arrival times" << endl;
    for (int i = 0; i < n; i++)
        cin >> processes[i].arrival_time;

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

    int time_frame = 0;
    for (int i = 0; i < n; i++)
    {
        time_frame += processes[i].burst_time;
```

```
        processes[i].turn_around_time = time_frame -
processes[i].arrival_time;

        if (i == 0)
            processes[i].waiting_time = 0;
        else
            processes[i].waiting_time =
processes[i].turn_around_time - processes[i].burst_time;
    }

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

    float avg_wt = 0.0;
    float avg_tat = 0.0;

    float sum_wt = 0.0;
    float sum_tat = 0.0;

    cout << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "Process " << processes[i].process_id <<
": ";
        cout << "Waiting Time: " <<
processes[i].waiting_time << " \t| ";
        cout << "Turnaround Time: " <<
processes[i].turn_around_time << endl;

        sum_wt += processes[i].waiting_time;
        sum_tat += processes[i].turn_around_time;
    }

    avg_wt = sum_wt / (float)n;
    avg_tat = sum_tat / (float)n;

    cout << endl;

    cout << "Average Waiting Time: \t\t";
```

```
    cout << fixed << setprecision(2) << avg_wt;
    cout << " time units" << endl;

    cout << "Average Turnaround Time: \t";
    cout << fixed << setprecision(2) << avg_tat;
    cout << " time units" << endl;
    return 0;
}
```

Analyzing the code:

The code begins by declaring the required built-in libraries namely `iostream`, `vector`, `algorithm`, and `iomanip`. The `iostream` provides the input and output mechanisms `cin` and `cout`, and the `vector` header file provides the `vector` class. Additionally, we need the `sort` function from the `algorithm` library to arrange the process queue by their ID or their arrival time when necessary. The `iomanip` provides `setprecision` function to display the floating point numbers in the output.

The details of each process is declared in a `struct`:

```
struct process
{
    int process_id = 0;
    int burst_time = 0;
    int arrival_time = 0;
    int completion_time = 0;
    int waiting_time = 0;
    int turn_around_time = 0;
};
```

Then, in the `main` function, these processes are stored in a vector array called `processes`:

```
vector<process> processes = {};
```

This is done because we need a mechanism to link process information, such as burst time and arrival time, when sorting them. Storing the information in a `struct` ensures the processes are isolated from each other, and the `vector` array ensures all of the process information can be sorted at once.

We sort the processes either by their process ID or their arrival time using the functions below:

```
bool sortByArrival(const process &p1, const process &p2)
{
    return p1.arrival_time < p2.arrival_time;
}

bool sortByID(const process &p1, const process &p2)
{
    return p1.process_id < p2.process_id;
}
```

In the main function, we take user input and store the number of processes, their CPU time and arrival time, like below. Here we also sort them by their arrival times:

```
int n;
vector<process> processes = {};

cout << "Enter the number of processes: ";
cin >> n;
for (int i = 0; i < n; i++)
    processes.push_back({i + 1});

cout << endl;
cout << "Enter the CPU times" << endl;
```

```
for (int i = 0; i < n; i++)
    cin >> processes[i].burst_time;

cout << endl;
cout << "Enter the arrival times" << endl;
for (int i = 0; i < n; i++)
    cin >> processes[i].arrival_time;

sort(processes.begin(), processes.end(), sortByArrival);
```

We declare a variable `time_frame` to keep track of the elapsed time since the processes started running. Then we use appropriate formulas to calculate their Waiting Time (**WT**) and Turnaround Time (**TAT**). The process that came at the zeroth time unit has no waiting time. So, $\mathbf{WT_{P1} = 0}$

```
int time_frame = 0;
for (int i = 0; i < n; i++)
{
    time_frame += processes[i].burst_time;
    processes[i].turn_around_time = time_frame -
processes[i].arrival_time;

    if (i == 0)
        processes[i].waiting_time = 0;
    else
        processes[i].waiting_time =
processes[i].turn_around_time - processes[i].burst_time;
}
```

Then we sort the processes by their ID and display the results:

```
sort(processes.begin(), processes.end(), sortByID);

float avg_wt = 0.0;
float avg_tat = 0.0;
```

```
float sum_wt = 0.0;
float sum_tat = 0.0;

cout << endl;
for (int i = 0; i < n; i++)
{
    cout << "Process " << processes[i].process_id << ": ";
    cout << "Waiting Time: " << processes[i].waiting_time
<< " \t| ";
    cout << "Turnaround Time: " <<
processes[i].turn_around_time << endl;

    sum_wt += processes[i].waiting_time;
    sum_tat += processes[i].turn_around_time;
}

avg_wt = sum_wt / (float)n;
avg_tat = sum_tat / (float)n;

cout << endl;

cout << "Average Waiting Time: \t\t";
cout << fixed << setprecision(2) << avg_wt;
cout << " time units" << endl;

cout << "Average Turnaround Time: \t";
cout << fixed << setprecision(2) << avg_tat;
cout << " time units" << endl;
```

Output:

```
  Enter the number of processes:
+ 3

  Enter the CPU times
+ 5 7 9
```

```
  Enter the arrival times
+ 4 0 2

  Process 1: Waiting Time: 12 | Turnaround Time: 17
  Process 2: Waiting Time: 0  | Turnaround Time: 7
  Process 3: Waiting Time: 5  | Turnaround Time: 14

  Average Waiting Time:    5.67 time units
  Average Turnaround Time: 12.67 time units
```

Challenges I faced while writing the code:

At first, I thought of using three arrays to store the process ID, CPU time, and arrival time of the processes. However, I came to realize that if we use separate arrays, we would not be able to sort all their related information together. We needed a method to group the process information together, so I used a `vector` array and isolated each process in a `struct`.

Shortest Job First (SJF)

Shortest Job First (**SJF**) is a scheduling algorithm that selects the waiting process with the smallest next CPU burst time to be executed next. Its primary goal is to minimize the average waiting time for all processes by prioritizing those that require the least execution time.

Here is my implementation of the **SJF** algorithm in **C++**:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct process
{
    int process_id = 0;
    int arrival_time = 0;
    int burst_time = 0;
    int completion_time = 0;
    int turn_around_time = 0;
    int waiting_time = 0;
    bool completed = 0;
};

bool sortByArrival(const process &p1, const process &p2)
{
    return p1.arrival_time < p2.arrival_time;
}

bool sortByID(const process &p1, const process &p2)
{
    return p1.process_id < p2.process_id;
}
```

```
int main()
{
    int n;
    vector<process> processes = {};

    cout << "Enter the number of processes: ";
    cin >> n;

    for (int i = 0; i < n; i++)
    {
        processes.push_back({i + 1});
        processes[i].completed = false;
    }

    cout << endl;
    cout << "Enter the CPU times" << endl;
    for (int i = 0; i < n; i++)
        cin >> processes[i].burst_time;

    cout << endl;
    cout << "Enter the arrival times" << endl;
    for (int i = 0; i < n; i++)
        cin >> processes[i].arrival_time;

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

    int timer = 0;
    int completed_processes = 0;
    float sum_wt = 0;
    float sum_tat = 0;

    while (completed_processes < n)
    {
        int min_job_index = -1;
        int min_bt = INT_MAX;

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

```
            if (processes[i].arrival_time <= timer && !
processes[i].completed)
            {
                if (processes[i].burst_time < min_bt)
                {
                    min_bt = processes[i].burst_time;
                    min_job_index = i;
                }
            }

        if (min_job_index != -1)
        {
            process &curr_process =
processes[min_job_index];
            curr_process.completion_time = timer +
curr_process.burst_time;
            curr_process.turn_around_time =
curr_process.completion_time - curr_process.arrival_time;
            curr_process.waiting_time =
curr_process.turn_around_time - curr_process.burst_time;

            timer = curr_process.completion_time;
            curr_process.completed = true;
            completed_processes++;
        }
        else
            timer++;
    }

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

    cout << endl;
    for (int i = 0; i < n; i++)
    {
        cout << "Process " << processes[i].process_id <<
": ";
        cout << "Waiting Time: " <<
processes[i].waiting_time << " \t| ";
```

```
        cout << "Turnaround Time: " <<
processes[i].turn_around_time << endl;

        sum_wt += processes[i].waiting_time;
        sum_tat += processes[i].turn_around_time;
    }

    float avg_wt = 0.0;
    float avg_tat = 0.0;
    avg_wt = sum_wt / (float)n;
    avg_tat = sum_tat / (float)n;

    cout << endl;

    cout << "Average Waiting Time: \t\t";
    cout << fixed << setprecision(2) << avg_wt;
    cout << " time units" << endl;

    cout << "Average Turnaround Time: \t";
    cout << fixed << setprecision(2) << avg_tat;
    cout << " time units" << endl;
    return 0;
}
```

Analyzing the code:

Here, the code implementation is very similar to the **FCFS** implementation. We use all the same libraries mentioned in the **FCFS** code and take user input in a similar manner. The main difference comes in the SJF logic itself.

Unlike **FCFS**, we have an integer `timer` that keeps track of CPU time and another integer `completed_processes` that counts how many processes have finished. Both are initially `0` :

```
int timer = 0;
int completed_processes = 0;
```

The following while loop runs until all processes are complete. This finds the shortest burst time among all **arrived** and **"yet to be completed"** processes:

```
while (completed_processes < n)
{
    int min_job_index = -1;
    int min_bt = INT_MAX;

    for (int i = 0; i < n; ++i)
        if (processes[i].arrival_time ⩽ timer && !
processes[i].completed)
        {
            if (processes[i].burst_time < min_bt)
            {
                min_bt = processes[i].burst_time;
                min_job_index = i;
            }
        }
    // ...
}
```

If such process exists, then we calculate their **WT** and **TAT** and set their `completed` flag to `true`. We also increment the `completed_processes` by 1:

```
if (min_job_index ≠ -1)
{
    process &curr_process = processes[min_job_index];
    curr_process.completion_time = timer +
curr_process.burst_time;
```

```
    curr_process.turn_around_time =
curr_process.completion_time - curr_process.arrival_time;
    curr_process.waiting_time =
curr_process.turn_around_time - curr_process.burst_time;

    timer = curr_process.completion_time;
    curr_process.completed = true;
    completed_processes++;
}
else
    timer++;
```

Otherwise, if no process has arrived yet, `timer` is incremented by 1 (**CPU is idle**).

Then we sort the processes by their ID and display the results:

```
sort(processes.begin(), processes.end(), sortByID);

cout << endl;
for (int i = 0; i < n; i++)
{
    cout << "Process " << processes[i].process_id << ": ";
    cout << "Waiting Time: " << processes[i].waiting_time
<< " \t| ";
    cout << "Turnaround Time: " <<
processes[i].turn_around_time << endl;

    sum_wt += processes[i].waiting_time;
    sum_tat += processes[i].turn_around_time;
}

float avg_wt = 0.0;
float avg_tat = 0.0;
avg_wt = sum_wt / (float)n;
avg_tat = sum_tat / (float)n;

cout << endl;
```

```
cout << "Average Waiting Time: \t\t";
cout << fixed << setprecision(2) << avg_wt;
cout << " time units" << endl;

cout << "Average Turnaround Time: \t";
cout << fixed << setprecision(2) << avg_tat;
cout << " time units" << endl;
```

Output:

```
  Enter the number of processes:
+ 3

  Enter the CPU times
+ 5 7 9

  Enter the arrival times
+ 4 0 2

Process 1: Waiting Time: 3  | Turnaround Time: 8
Process 2: Waiting Time: 0  | Turnaround Time: 7
Process 3: Waiting Time: 10 | Turnaround Time: 19

Average Waiting Time:    4.33 time units
Average Turnaround Time: 11.33 time units
```

Challenges I faced while writing the code:

I found it tricky to keep track of the shortest job, still after a bit of tinkering, I stored the shortest burst time and its corresponding process index in two variables: `min_job_index` and `min_bt`. I also added a `completed` flag to keep track of completed processes, so I can exclude them while looking for the shortest job in another iteration. It also took a few attempts to get it right.

Shortest Remaining Time First (SRTF)

The Shortest Remaining Time First (**SRTF**) is a scheduling algorithm where the process with the smallest remaining execution time is always chosen to run next. It is the preemptive version of Shortest Job First (**SJF**) algorithm.

Here is my implementation of the **SRTF** algorithm in **C++**:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct process
{
    int process_id = 0;
    int burst_time = 0;
    int arrival_time = 0;
    int completion_time = 0;
    int waiting_time = 0;
    int turn_around_time = 0;
    int remaining_time = 0;
};

bool sortByID(const process &p1, const process &p2)
{
    return p1.process_id < p2.process_id;
}

int main()
{
    int n;
    vector<process> processes = {};

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

```
    cin >> n;
    for (int i = 0; i < n; i++)
        processes.push_back({i + 1});

    cout << endl;
    cout << "Enter the CPU times" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> processes[i].burst_time;
        processes[i].remaining_time =
processes[i].burst_time;
    }

    cout << endl;
    cout << "Enter the arrival times" << endl;
    for (int i = 0; i < n; i++)
        cin >> processes[i].arrival_time;

    int timer = 0;
    int completed_processes = 0;
    int shortest_job_index = -1;
    int min_remaining_time = INT_MAX;
    bool process_running = false;

    while (completed_processes < n)
    {
        min_remaining_time = INT_MAX;
        shortest_job_index = -1;
        process_running = false;

        for (int i = 0; i < n; ++i)
            if (processes[i].arrival_time <= timer &&
processes[i].remaining_time > 0)
            {
                if (processes[i].remaining_time <
min_remaining_time)
                {
                    min_remaining_time =
```

```
processes[i].remaining_time;
                    shortest_job_index = i;
                    process_running = true;
                }
            }

        if (!process_running)
        {
            timer++;
            continue;
        }

        process &curr_process =
processes[shortest_job_index];
        curr_process.remaining_time--;
        timer++;

        if (curr_process.remaining_time == 0)
        {
            completed_processes++;
            curr_process.completion_time = timer;
            curr_process.turn_around_time =
curr_process.completion_time - curr_process.arrival_time;
            curr_process.waiting_time =
curr_process.turn_around_time - curr_process.burst_time;
        }
    }

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

    float sum_wt = 0.0;
    float sum_tat = 0.0;
    float avg_wt = 0.0;
    float avg_tat = 0.0;

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

```
        cout << "Process " << processes[i].process_id <<
": ";
        cout << "Waiting Time: " <<
processes[i].waiting_time << " \t| ";
        cout << "Turnaround Time: " <<
processes[i].turn_around_time << endl;

        sum_wt += processes[i].waiting_time;
        sum_tat += processes[i].turn_around_time;
    }

    avg_wt = sum_wt / (float)n;
    avg_tat = sum_tat / (float)n;

    cout << endl;

    cout << "Average Waiting Time: \t\t";
    cout << fixed << setprecision(2) << avg_wt;
    cout << " time units" << endl;

    cout << "Average Turnaround Time: \t";
    cout << fixed << setprecision(2) << avg_tat;
    cout << " time units" << endl;
    return 0;
}
```

Analyzing the code:

Since it is the **preemptive** version of **SJF**, some of the implementation is similar to **SJF**. But there are some noticible differences.

In the **SRTF** implementation, instead of using a `completed` flag like **SJF**, each process tracks how much CPU time remains:

```
int remaining_time = 0;
```

When a process is chosen for execution, its `remaining_time` is decremented by 1 for each time unit:

```
curr_process.remaining_time--;
```

When the `remaining_time` reaches zero, the process is considered "finished":

```
if (curr_process.remaining_time == 0)
{
    completed_processes++;
    curr_process.completion_time = timer;
}
```

In the non-preemptive SJF, the CPU runs the chosen process for its entire burst time before checking for the next one. Once a process is chosen, the timer jumps forward by the process's burst duration. This means the CPU makes scheduling decisions only when a process completes; not in between.

```
curr_process.completion_time = timer +
curr_process.burst_time;
timer = curr_process.completion_time;
```

In contrast, the preemptive version increases the timer one unit at a time:

```
timer++;
```

and re-evaluates the remaining times of all arrived processes at each tick. This enables our algorithm to check continuously whether a shorter job has appeared:

```
for (int i = 0; i < n; ++i)
    if (processes[i].arrival_time ⩽ timer &&
processes[i].remaining_time > 0)
    {
        if (processes[i].remaining_time <
min_remaining_time)
        {
            min_remaining_time =
processes[i].remaining_time;
            shortest_job_index = i;
        }
    }
```

In the non-preemptive version, the code searches for the shortest job only among the processes that have already arrived and are not completed:

```
if (processes[i].arrival_time ⩽ timer && !
processes[i].completed)
{
    if (processes[i].burst_time < min_bt)
    {
        min_bt = processes[i].burst_time;
        min_job_index = i;
    }
}
```

Once a process is selected (meaning `min_job_index ≠ -1`), it runs until completion, and only then does the CPU look for another job.

However, in the preemptive version, the logic is similar but continuously dynamic. At every single time unit, the scheduler looks for the process with the shortest remaining time, not total burst time:

```
if (processes[i].remaining_time < min_remaining_time)
{
    min_remaining_time = processes[i].remaining_time;
    shortest_job_index = i;
}
```

As this check runs in every iteration of the main while loop (which increments `timer` by one each time), the algorithm can instantly respond to new processes as they arrive, possibly interrupting the currently running one.

Output:

```
  Enter the number of processes:
+ 3

  Enter the CPU times
+ 5 7 9

  Enter the arrival times
+ 4 0 2

Process 1: Waiting Time: 3  | Turnaround Time: 8
Process 2: Waiting Time: 0  | Turnaround Time: 7
Process 3: Waiting Time: 10 | Turnaround Time: 19

Average Waiting Time:    4.33 time units
Average Turnaround Time: 11.33 time units
```

Challenges I faced while writing the code:

Since the code is another implementation of the existing **SJF** algorithm, most of the code was similar. However, I struggled a lot to understand when to increment

the `timer` and how to select the next process based on its remaining time. It took a few attempts, but eventually, I got it right.
