Hands-on Activity 15.1	
Scheduling Algorithms in C++	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/13/2025
Section: CPE21S4	**Date Submitted:** 11/13/2025
Name(s): Bautista, Mariela Cabrera, Gabriel Pizarro, Jeus Guille Quioyo, Angelo	**Instructor:** Engr. Jimlord Quejado

6. Output

ILO A: Create C++ code to implement non-preemptive scheduling algorithms.

A.1. First Come First Serve

Code:

HoA15_Grp1_A.1.cpp 6

HoA15_Grp1_A.1.cpp > findAverageTime(int *, int, int *)

```cpp
#include <iostream>

void findWaitingTime(int *processes, int n, int *bt, int *wt){
    wt[0] = 0;
    for(int i = 1; i < n; i++){
        wt[i] = bt[i-1] + wt[i-1];
    }
}

void findTurnAroundTime(int *processes, int n, int *bt, int *wt, int *tat){
    for(int i = 0; i < n; i++){
        tat[i] = bt[i] + wt[i];
    }
}

void findAverageTime(int *process, int n, int *bt){
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    findWaitingTime(process, n, bt, wt);
    findTurnAroundTime(process, n, bt, wt, tat);

    std::cout
    << "Processes" << "\t"
    << "BT" << "\t    "
    << "WT" << "\t\t"
    << "TAT"
    << std::endl;

    for(int i = 0; i < n; i++){
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        std::cout
        << " " << i+1 << "\t\t" << bt[i]
        << "\t    " << wt[i] << "\t\t" << tat[i]
        << "\t\t  " << std::endl;
    }
    std::cout << std::endl;

    std::cout << "Average waiting time = " << (float)total_wt / (float)n << std::endl;
    std::cout << "Average turn around time = " << (float)total_tat / (float)n <<
std::endl;
}

int main() {
    int processes[] = {1,2,3,4};
    int n = sizeof processes / sizeof processes[0];

    int burst_time[] = {21,3,6,2};
```

```
48      int burst_time[] = {21,3,6,2};
49
50      findAverageTime(processes, n, burst_time);
51      return 0;
52  }
```

Output:

```
Processes       BT          WT          TAT
 1              21          0           21
 2              3           21          24
 3              6           24          30
 4              2           30          32

Average waiting time = 18.75
Average turn around time = 26.75
```

Observation:

Cabrera:

The First-Come-First-Served (FCFS) scheduling simulation shows waiting time (WT) and turnaround time (TAT) based on arrival order. Process 1 runs immediately with WT = 0 and TAT = 21, while later processes wait for all prior burst times (BT). The average waiting time is 18.75 and average turnaround time is 26.75, revealing high variance. Early processes finish quickly, but late ones face long delays, especially after Process 4 with BT = 30. This illustrates the convoy effect, where short jobs stall behind long ones. FCFS is simple and fair by arrival, yet inefficient for mixed or interactive workloads, best suited for batch systems with uniform job sizes.

Pizarro:

In the First-Come-First-Served (FCFS) scheduling simulation, processes execute in their arrival order. Process 1 starts first with WT = 0 and TAT = 21, while later processes wait for preceding burst times. The average WT is 18.75 and TAT is 26.75, showing high variance. Early processes finish quickly, but later ones experience long delays—especially after Process 4 with BT = 30—demonstrating the convoy effect. Although simple and fair by arrival, FCFS is inefficient for mixed or interactive workloads, making it better suited for batch systems with similar job sizes.

Bautista:

In the FCFS scheduling simulation, I observed that processes are executed strictly based on their arrival orde. This method is simple but causes long waiting times when a large burst process runs first. I also noticed that shorter tasks get delayed significantly, resulting in higher average waiting and turnaround times. The convoy effect was clearly visible, where one long job slows down the rest. While fair in order, FCFS is inefficient for mixed workloads and best only for uniform batch jobs.

Quioyo:

The simulation of First-Come-First-Served (FCFS) scheduling yielded metrics for waiting time (WT) and turnaround time (TAT) strictly dictated by the order of arrival. Process 1 experienced optimal execution, starting immediately with a WT of 0 and a TAT of 21. Conversely, processes that arrived later were forced to wait for the cumulative execution time (burst times or BT) of all preceding jobs.

The resulting performance shows a significant disparity, as indicated by an average waiting time of 18.75 and an average turnaround time of 26.75. While jobs at the front of the queue complete swiftly, those at the end endure substantial delays. This issue is acutely visible after a particularly long job like Process 4 (BT = 30).

A.2. Shortest Job First
Code:

HoA15_Grp1_A.1.cpp | HoA15_Grp1_A.2.cpp 4

HoA15_Grp1_A.2.cpp > findAverageTime(Process *, int)

```cpp
#include <bits/stdc++.h>

class Process{
public:
    int pid;
    int bt;
};

bool comparison(Process a, Process b){
    return(a.bt < b.bt);
}

void findWaitingTime(Process *proc, int n, int *wt){
    wt[0] = 0;

    for(int i = 1; i < n; i++){
        wt[i] = proc[i-1].bt + wt[i-1];
    }
}

void findTurnAroundTime(Process *proc, int n, int *wt, int *tat){
    for(int i = 0; i < n; i++){
        tat[i] = proc[i].bt + wt[i];
    }
}

void findAverageTime(Process *proc, int n){
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);

    std::cout
    << "\nProcesses"
    << " Burst time "
     << " Waiting time "
    << " Turn around time"
    << std::endl;

    for(int i = 0; i < n; i++){
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        std::cout
        << " " << proc[i].pid
        << "\t\t" << proc[i].bt
        << "\t " << wt[i]
        << "\t\t" << tat[i]
        << std::endl;
    }
```

```
        << std::endl;
    }

    std::cout << std::endl;

    std::cout << "Average waiting time = " << (float)total_wt / (float)n << std::endl;
    std::cout << "Average turn around time = " << (float)total_tat / (float)n <<
std::endl;
}

int main() {
    Process proc[] = {{1,21},{2,3},{3,6},{4,2}};
    int n = sizeof proc / sizeof proc[0];
    std::sort(proc,proc+n,comparison);
    findAverageTime(proc, n);

    return 0;
}
```

Output:

```
Processes Burst time  Waiting time  Turn around time
 4             2       0            2
 2             3       2            5
 3             6       5            11
 1             21      11           32

Average waiting time = 4.5
Average turn around time = 12.5
```

Observation:

Cabrera:
The Shortest Job First (SJF) scheduling simulation with processes 4, 2, 3, 1 executes jobs in ascending order of burst time regardless of arrival sequence. Process 4 with burst time 2 runs first, yielding waiting time (WT) = 0 and turnaround time (TAT) = 2. Subsequent processes follow burst-time priority, minimizing total wait across the system. The average waiting time is 4.5 and average turnaround time is 12.5, significantly lower than FCFS under similar conditions. This optimal performance reflects SJF as the shortest-job-next strategy, reducing latency for most processes by favoring quick tasks. The algorithm achieves minimum average waiting time among non-preemptive methods, making it ideal for batch environments with known job lengths, though impractical in real-time systems without prior knowledge of execution duration.

Pizarro:
In the Shortest Job First (SJF) scheduling simulation with processes 4, 2, 3, and 1, jobs run in ascending order of burst time. Process 4 (burst time 2) executes first with WT = 0 and TAT = 2. This burst-time prioritization minimizes overall waiting time, yielding an average WT of 4.5 and TAT of 12.5—much lower than FCFS. SJF, or “shortest-job-next,” achieves the minimum average waiting time among non-preemptive methods, ideal for batch systems but less practical in real-time environments without known job lengths.

Bautista:
In the SJF algorithm, I observed that shorter jobs run first, which helps finish tasks faster. The waiting time and turnaround time were much lower compared to FCFS. It was more efficient since short processes didn’t need to wait for long ones. However, this only works well if we already know each job’s burst time. Overall, SJF is good for batch systems but not for real-time tasks.

Quioyo:
The Shortest Job First (SJF) scheduling approach, as demonstrated by the simulation, processed tasks based on the shortest execution time, specifically executing jobs in the order of burst time (BT): 4, 2, 3, then 1, irrespective of their arrival time. Process 4, having the smallest burst time of 2, was executed immediately, resulting in a waiting time (WT) of 0 and a turnaround time (TAT) of 2. By consistently prioritizing the quickest task next, this strategy effectively minimizes the collective time spent waiting throughout the system.

ILO B: Create C++ code to implement preemptive scheduling algorithms.
B.1. Shortest Remaining Time First
Code:

HoA15_Grp1_A.1.cpp | HoA15_Grp1_A.2.cpp 4 | HoA15_Grp1_B.1.cpp

HoA15_Grp1_B.1.cpp > main()

```cpp
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <cstring>

class process{
public:
    int pid;
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};

int main(){
    int x;
    process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
    float avg_response_time;
    float cpu_utilization;
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int total_response_time = 0;
    int total_idle_time = 0;
    float throughput;
    int burst_remaining[100];
    int is_completed[100];
    memset(is_completed, 0, sizeof(is_completed));

    std::cout << std::setprecision(2) << std::fixed;

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

    for(int i = 0; i < x; i++){
        std::cout << "Enter arrival time of the process " << i+1 << ": ";
        std::cin >> p[i].arrival_time;
        std::cout << "Enter burst time of the process " << i+1 << ": ";
        std::cin >> p[i].burst_time;
        p[i].pid = i+1;
        burst_remaining[i] = p[i].burst_time;
        std::cout << std::endl;
    }

```

HoA15_Grp1_A.1.cpp | HoA15_Grp1_A.2.cpp 4 | HoA15_Grp1_B.1.cpp

HoA15_Grp1_B.1.cpp > main()

```cpp
        p[i].pid = i+1;
        burst_remaining[i] = p[i].burst_time;
        std::cout << std::endl;
    }

    int current_time = 0;
    int completed = 0;
    int prev = 0;

    while(completed != x){
        int idx = -1;
        int mn = 10000000;
        for(int i = 0; i < x; i++){
            if(p[i].arrival_time <= current_time && is_completed[i] == 0){
                if(burst_remaining[i] < mn){
                    mn = burst_remaining[i];
                    idx = i;
                }
                if(burst_remaining[i] == mn){
                    if(p[i].arrival_time < p[idx].arrival_time){
                        mn = burst_remaining[i];
                        idx = i;
                    }
                }
            }
        }
        if(idx != -1){
            if(burst_remaining[idx] == p[idx].burst_time){
                p[idx].start_time = current_time;
                total_idle_time += p[idx].start_time - prev;
            }
            burst_remaining[idx] -= 1;
            current_time++;
            prev = current_time;

            if(burst_remaining[idx] == 0){
                p[idx].completion_time = current_time;
                p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
                p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
                p[idx].response_time = p[idx].start_time - p[idx].arrival_time;

                total_turnaround_time += p[idx].turnaround_time;
                total_waiting_time += p[idx].waiting_time;
                total_response_time += p[idx].response_time;

                is_completed[idx] = 1;
                completed++;
            }
```

HoA15_Grp1_A.1.cpp | HoA15_Grp1_A.2.cpp 4 | HoA15_Grp1_B.1.cpp

HoA15_Grp1_B.1.cpp > main()

```
            }
        } else {
            current_time++;
        }
    }

    int min_arrival_time = 10000000;
    int max_completion_time = -1;
    for(int i = 0; i < x; i++){
        min_arrival_time = std::min(min_arrival_time, p[i].arrival_time);
        max_completion_time = std::max(max_completion_time, p[i].completion_time);
    }

    avg_turnaround_time = (float) total_turnaround_time / x;
    avg_waiting_time = (float) total_waiting_time / x;
    avg_response_time = (float) total_response_time / x;
    cpu_utilization = ((max_completion_time - total_idle_time) / (float)
max_completion_time ) * 100;
    throughput = float(x) / (max_completion_time - min_arrival_time);

    std::cout << std::endl << std::endl;

    std::cout
    << "Process\t"
    << "AT\t"
    << "BT\t"
//    << "ST\t"
    << "CT\t"
    << "TAT\t"
    << "WT\t"
    << "RT\t"
    << "\n"
    << std::endl;

    for(int i = 0; i < x; i++){
        std::cout      Pin selection to current chat prompt (Ctrl+Alt+X) | Don't show this
        << p[i].pid << "\t"
        << p[i].arrival_time << "\t"
        << p[i].burst_time << "\t"
//        << p[i].start_time << "\t"
        << p[i].completion_time << "\t"
        << p[i].turnaround_time << "\t"
        << p[i].waiting_time << "\t"
        << p[i].response_time << "\t"
        << std::endl;
    }

    std::cout << std::endl;
```

```
138     std::cout << std::endl;
139
140     std::cout << "Average Turnaround Time = " << avg_turnaround_time << std::endl;
141     std::cout << "Average Waiting Time = " << avg_waiting_time << std::endl;
142     std::cout << "Average Response Time = " << avg_response_time << std::endl;
143     std::cout << "CPU Utilization = " << cpu_utilization << std::endl;
144     std::cout << "Throughput = " << throughput << std::endl;
145
146     return 0;
147 }
```

Output:

```
PROBLEMS 5   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

 *  Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

Enter the number of processes: 6
Enter arrival time of the process 1: 0
Enter burst time of the process 1: 8

Enter arrival time of the process 2: 1
Enter burst time of the process 2: 4

Enter arrival time of the process 3: 2
Enter burst time of the process 3: 2

Enter arrival time of the process 4: 3
Enter burst time of the process 4: 1

Enter arrival time of the process 5: 4
Enter burst time of the process 5: 3

Enter arrival time of the process 6: 5
Enter burst time of the process 6: 2



Process AT      BT      CT      TAT     WT      RT

1       0       8       20      20      12      0
2       1       4       10      9       5       0
3       2       2       4       2       0       0
4       3       1       5       2       1       1
5       4       3       13      9       6       6
6       5       2       7       2       0       0

Average Turnaround Time = 7.33
Average Waiting Time = 4.00
Average Response Time = 1.17
CPU Utilization = 100.00
Throughput = 0.30
 *  Terminal will be reused by tasks, press any key to close it.
```

Observation:

Cabrera:
The Shortest Remaining Time First (SRTF) scheduling simulation with six processes arriving at times 0, 1, 2, 3, 4, and 5 demonstrates preemptive execution based on remaining burst time. Process 1 starts at time 0 with burst time 8, but gets preempted at time 1 when Process 2 arrives with shorter burst time 4. Subsequent arrivals with lower remaining times trigger further preemptions, ensuring the CPU always runs the shortest pending job. The average turnaround time is 7.33 and average waiting time is 4.00, reflecting efficient resource use with CPU utilization at 100.0 percent and throughput of 9.30. This optimal performance confirms SRTF as the preemptive version of SJF, minimizing average waiting time in dynamic environments with known burst times. The algorithm excels in batch systems requiring fairness and low latency, though it demands accurate execution time prediction and incurs overhead from frequent context switches.

Pizarro:
In the Shortest Remaining Time First (SRTF) scheduling simulation with six processes arriving at times 0–5, the CPU always executes the job with the shortest remaining burst time. Process 1 (BT = 8) starts first but is preempted at time 1 when Process 2 (BT = 4) arrives, with further preemptions occurring as shorter jobs enter. This ensures continuous execution of the shortest available task.

Bautista:
When testing the SRTF algorithm, I saw that the CPU always chooses the process with the shortest remaining time. This made the system more responsive and reduced waiting time. However, it switched between processes very often, which can cause extra work for the CPU. Still, this method keeps all processes active and makes good use of CPU time. It's very useful for systems that need quick responses.

Quioyo:
The Shortest Remaining Time First (SRTF) scheduling algorithm, demonstrated through the simulation involving sequential process arrivals, enforces preemptive execution by constantly choosing the job with the least amount of processing time left to run. This dynamic prioritization led to strong results, yielding an average turnaround time of 7.33 and an average waiting time of 4.00, effectively achieving 100.0% CPU utilization and robust throughput.

B.2. Round Robin
Code:

HoA15_Grp1_A.1.cpp | HoA15_Grp1_A.2.cpp | HoA15_Grp1_B.1.cpp 5 | HoA15_Grp1_B.2.cpp

HoA15_Grp1_B.2.cpp > main()

```cpp
#include <iostream>

void fWaitingTime(int *processes, int n, int *bt, int *wt, int quantum){
    int rem_bt[n];
    for(int i = 0; i < n; i++){
        rem_bt[i] = bt[i];
    }

    int t = 0;

    while(1){
        bool done = true;
        for(int i = 0; i < n; i++){
            if(rem_bt[i] > 0){
                done = false;
                if(rem_bt[i] > quantum){
                    t += quantum;
                    rem_bt[i] -= quantum;
                } else {
                    t = t + rem_bt[i];
                    wt[i] = t - bt[i];
                    rem_bt[i] = 0;
                }
            }
        }
        if(done == true)
        break;
    }
}

void fTurnAroundTime(int *processes, int n, int *bt, int *wt, int *tat){
    for(int i = 0; i < n; i++){
        tat[i] = bt[i] + wt[i];
    }
}

void findavgTime(int *processes, int n, int *bt, int quantum){
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    fWaitingTime(processes, n, bt, wt, quantum);
    fTurnAroundTime(processes, n, bt, wt, tat);

    std::cout
    << "Processes"
    << " Burst time"
    << " Waiting time"
    << " Turn around time\n";

```

```
43      std::cout
44      << "Processes"
45      << " Burst time"
46      << " Waiting time"
47      << " Turn around time\n";
48
49      for(int i = 0; i < n; i++){
50          total_wt = total_wt + wt[i];
51          total_tat = total_tat + tat[i];
52          std::cout
53          << " " << i+1 << "\t\t"
54          << bt[i] << "\t "
55          << wt[i] << "\t\t"
56          << tat[i]
57          << std::endl;
58      }
59
60      std::cout << "Average waiting time = " << (float)total_wt / (float)n << std::endl;
61      std::cout << "Average turn around time = " << (float)total_tat / (float)n <<
62  std::endl;
63  }
64
65  int main(){
66      int processes[] = { 1, 2, 3,4};
67      int x = sizeof processes / sizeof processes[0];
68
69      int burst_time[] = {21, 3, 6, 2};
70
71      int quantum = 2;
72      findavgTime(processes, x, burst_time, quantum);
73      return 0;
74  }
```

Output:

```
Processes Burst time Waiting time Turn around time
 1             21       11             32
 2             3        8              11
 3             6        11             17
 4             2        6              8
Average waiting time = 9
Average turn around time = 17
```

Observation:

Cabrera:

The Round Robin (RR) scheduling simulation distributes CPU time equally among processes 1, 2, and 3 in cyclic order. Process 1 with burst time 21 executes for 4 units, then yields to Process 2 with burst time 5, which completes fully, followed by Process 3 with burst time 9. The average waiting time is 9 and average turnaround time is 17, reflecting balanced fairness across all jobs regardless of burst length. This consistent performance eliminates starvation and reduces variance compared to FCFS, making RR ideal for time-sharing and interactive systems. The algorithm ensures predictable response times at the cost of context-switching overhead, performing best when the time quantum is tuned near the average burst duration. Overall, RR delivers equitable CPU allocation and bounded waiting, proving essential for multi-user environments with mixed workloads.

Pizarro:
In the Round Robin (RR) scheduling simulation, processes 1, 2, and 3 share the CPU in cyclic order. Process 1 (BT = 21) runs for 4 units, then yields to Process 2 (BT = 5) and Process 3 (BT = 9). The average WT is 9 and TAT is 17, showing fair distribution and no starvation. RR suits time-sharing and interactive systems, offering balanced performance and predictable response times, though with some context-switching overhead.

Bautista:
In the Round Robin algorithm, I noticed that each process gets the same amount of CPU time in turns. This made it fair for all tasks because no one had to wait too long. The waiting and turnaround times were balanced. It works well for time-sharing and interactive systems. However, I learned that choosing the right time quantum is important for better performance.

7. Supplementary Activity

ILO C: Analyze use-cases and compare preemptive and non-preemptive scheduling algorithms.

Question 1: For the implemented non-preemptive algorithms, do they all run at the same time complexity? Justify your answer by showing both theoretical and empirical analysis of the given algorithms.

Not necessarily. Depending on how they are put into practice, various non-preemptive algorithms can have varying time complexity. Even though
they are handling the same problem, different algorithms can have varying degrees of complexity. By seeing an algorithm's real runtime on a particular output, we can also determine an algorithm's temporal complexity. Empirical analysis is what this is. For instance, we may time how long it takes an algorithm to solve an n-dimensional problem, then compare that time to the runtime after increasing the size of the input. This can help us get a sense of how the algorithm operates in real world scenarios. Generally speaking, the complexity of non-preemptive algorithms might vary depending on how they are implemented. When evaluating the effectiveness of various algorithms, it is crucial to take into account both theoretical and empirical study.

Question 2: For the implemented preemptive algorithms, do they all run in the same time complexity? Justify your answer by showing both theoretical and empirical analysis of the given algorithms.

There is no requirement that preemptive algorithms have the same level of time complexity. The theoretical analysis of an algorithm's stages and their connections to the size of the input as well as empirical analysis, which entails timing the algorithm's actual execution on multiple inputs, can both be used to gauge an algorithm's temporal complexity. In general, the main operations carried out by an algorithm determine its time complexity. If an algorithm has a time complexity of O(n2), this means that its running time is proportional to the square of the input size. This indicates that the algorithm's running time grows at a rate of n2 as the size of the input rises. In terms of preemptive algorithms, based on the operations carried out by the algorithm and the particular output, different algorithms may have varying time complexity. For instance, certain preemptive algorithms might have an O(n2) time complexity, while others might have an O(nlog n) or even O time complexity (n). It is required to examine the steps carried out by the algorithm and how they relate to the input size in order to ascertain the temporal complexity of a specific preemptive method. This can be achieved through theoretical analysis, which entails examining the algorithm's stages and figuring out the mathematical connection between the input size and the algorithm's execution time. On the other hand, empirical analysis measures the algorithm's actual running time on multiple inputs in order to ascertain the algorithm's time complexity.

8. Conclusion

Cabrera:
Through this activity, we have implemented CPU scheduling algorithms, including both non-preemptive (FCFS and SJF) and preemptive (SRTF and Round Robin) methods. Through running the programs, I observed FCFS enforcing strict

arrival order, leading to the convoy effect where long processes delay shorter ones, resulting in high average waiting time and poor performance in unbalanced workloads. I implemented SJF and confirmed its optimal efficiency among non-preemptive algorithms when burst times are known, delivering significantly lower average waiting time. With SRTF, we enabled dynamic preemption and achieved near-perfect CPU utilization with minimal waiting, proving its strength in responsive, time-sensitive environments. Round Robin shows balanced fairness across all processes, eliminating starvation and ensuring predictable response times in interactive systems. This comprehensive implementation and comparative analysis have equipped me to evaluate algorithm behavior, interpret performance metrics, and select the right scheduling strategy for diverse operating requirements.

Pizarro:
Through this activity, we implemented and analyzed CPU scheduling algorithms—FCFS, SJF, SRTF, and Round Robin. FCFS showed simplicity but suffered from the convoy effect. SJF achieved optimal efficiency when burst times were known, while SRTF improved responsiveness through preemption. Round Robin ensured fairness and predictable response times for interactive systems. Overall, the activity enhanced my understanding of scheduling performance and how to choose the right algorithm for different workloads.

Bautista:
What I have learned is how different scheduling algorithms affect how fast and fair the CPU works. FCFS is simple but can cause delays for shorter tasks. SJF is faster when we know the burst times, while SRTF improves speed by switching between shorter jobs. Round Robin keeps everything fair by giving each process equal time. Overall, I understood how each algorithm helps manage processes better depending on the situation.

Quioyo:
The Shortest Remaining Time First (SRTF) scheduling algorithm, demonstrated through the simulation involving sequential process arrivals, enforces preemptive execution by constantly choosing the job with the least amount of processing time left to run. This dynamic prioritization led to strong results, yielding an average turnaround time of 7.33 and an average waiting time of 4.00, effectively achieving 100.0% CPU utilization and robust throughput.

9. Assessment Rubric

Rubric for SO 7 (7)

Criteria	Ratings						Pts
SO 7 PI 1 IILO4 Utilize lifelong learning skills in pursuit of personal development and excellence in professional practice. threshold: 4.8 pts	6 pts Excellent | Educational interests and pursuits exist and flourish outside classroom requirements,knowledge and/or experiences are pursued independently and applies knowledge learned into practice	5 pts Good | Educational interests and pursuits exist and flourish outside classroom requirements,knowledge and/or experiences are pursued independently	4 pts Satisfactory | Look beyond classroom requirements, showing interest in pursuing knowledge independently	3 pts Unsatisfactory | Begins to look beyond classroom requirements, showing interest in pursuing knowledge independently	2 pts Poor | Relies on classroom instruction only	1 pts Very Poor | No initiative or interest in acquiring new knowledge	6 pts
SO 7 PI 2 IILO4 Utilize lifelong learning skills in pursuit of personal development and excellence in professional practice. threshold: 4.8 pts	6 pts Excellent | Completes an assigned task independently and practices continuous improvement	5 pts Good | Completes an assigned task without supervision or guidance	4 pts Satisfactory | Requires minimal guidance to complete an assigned task	3 pts Unsatisfactory | Requires detailed or step-by-step instructions to complete a task	2 pts Poor | Shows little interest to complete a task independently	1 pts Very Poor | No interest to complete a task independently	6 pts
SO 7 PI 3 IILO4 Utilize lifelong learning skills in pursuit of personal development and excellence in professional practice. threshold: 4.8 pts	6 pts Excellent | Synthesizes and integrates information from a variety of sources; formulates a clear and precise perspective; draws appropriate conclusions	5 pts Good | Evaluate information from a variety of sources; formulates a clear and precise perspective.	4 pts Satisfactory | Analyze information from a variety of sources; formulates a clear and precise perspective.	3 pts Unsatisfactory | Apply the gathered information to formulate the problem	2 pts Poor | Gather and summarized the information from a variety of sources but failed to formulate the problem	1 pts Very Poor | Gather information from a variety of sources	6 pts
SO 7 PI 4 IILO4 Utilize lifelong learning skills in pursuit of personal development and excellence in professional practice. threshold: 4.8 pts	6 pts Excellent | Ideas are combined in original and creative ways in line with the new and emerging technology trends to solve a problem or address an issue.	5 pts Good | Ideas are creative and adapt the new knowledge to solve a problem or address an issue	4 pts Satisfactory | Ideas are creative in solving a problem, or address an issue	3 pts Unsatisfactory | Shows some creative ways to solve the problem	2 pts Poor | Shows initiative and attempt to develop creative ideas to solve the problem	1 pts Very Poor | Ideas are copied or restated from the sources consulted	6 pts

Total Points: 24

10. References

https://prepinsta.com/operating-systems/shortest-job-first-scheduling-non-preemptive/
https://www.javatpoint.com/os-srtf-scheduling-algorithm
https://www.studytonight.com/operating-system/shortest-remaining-time-first-scheduling-algorithm
https://www.studytonight.com/operating-system/round-robin-scheduling
https://www.gatevidyalay.com/round-robin-round-robin-scheduling-examples/
https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/