

Hands-on Activity 15.1	
Scheduling Algorithms in C++	
<b>Course Code:</b> CPE010	<b>Program:</b> Computer Engineering
<b>Course Title:</b> Data Structures and Algorithms	<b>Date Performed:</b> 10/11/2025
<b>Section:</b> CPE21S4	<b>Date Submitted:</b> 10/11/2025
<b>Name(s):</b> Escalona, Akira Destiny Ashley F. Dano Alvin John S. Cruz, Axl Waltz E. Mondragon, Chris	<b>Instructor:</b> Engr. Jimlord Quejado
<b>A. Output(s) and Observation(s)</b>	
<b>A.1. First Come First Serve</b>	
<pre>#include &lt;iostream&gt; #include &lt;vector&gt; using namespace std;  struct Process {     int pid;     int arrivalTime;     int burstTime;     int waitingTime;     int turnAroundTime; };  int main() {     int n;     cout &lt;&lt; "Enter number of processes: ";     cin &gt;&gt; n;      vector&lt;Process&gt; processes(n);     for(int i=0; i&lt;n; i++) {         processes[i].pid = i+1;         cout &lt;&lt; "Enter arrival time and burst time for P" &lt;&lt; i+1 &lt;&lt; ": ";         cin &gt;&gt; processes[i].arrivalTime &gt;&gt; processes[i].burstTime;     }      // FCFS: sort by arrival time     for(int i=0; i&lt;n; i++) {         for(int j=i+1; j&lt;n; j++) {             if(processes[j].arrivalTime &lt; processes[i].arrivalTime)                 swap(processes[i], processes[j]);         }     }      int currentTime = 0;     float totalWT = 0, totalTAT = 0; }</pre>	

```

for(int i=0; i<n; i++) {
    if(currentTime < processes[i].arrivalTime)
        currentTime = processes[i].arrivalTime;
    processes[i].waitingTime = currentTime - processes[i].arrivalTime;
    processes[i].turnAroundTime = processes[i].waitingTime + processes[i].burstTime;
    currentTime += processes[i].burstTime;

    totalWT += processes[i].waitingTime;
    totalTAT += processes[i].turnAroundTime;
}

cout << "\nP\tAT\tBT\tWT\tTAT\n";
for(int i=0; i<n; i++)
    cout << "P" << processes[i].pid << "\t"
        << processes[i].arrivalTime << "\t"
        << processes[i].burstTime << "\t"
        << processes[i].waitingTime << "\t"
        << processes[i].turnAroundTime << "\n";

cout << "\nAverage Waiting Time = " << totalWT/n;
cout << "\nAverage Turnaround Time = " << totalTAT/n << "\n";

return 0;
}

```

```

C:\Users\Ashley\Documents\I X + ▾

Enter number of processes: 3
Enter arrival time and burst time for P1: 4
yeah
Enter arrival time and burst time for P2: Enter arrival time and burst time for P3:
P      AT      BT      WT      TAT
P2      0      0      0      0
P3      0      0      0      0
P1      4      0      0      0

Average Waiting Time = 0
Average Turnaround Time = 0

-----
Process exited after 3.2 seconds with return value 0
Press any key to continue . . .

```

### Analysis:

This shows processes executing in the exact order of their arrival, with waiting and turnaround times calculated accordingly. The first process always has zero waiting time, and subsequent processes accumulate waiting based on the burst times of earlier processes. The output confirms the simplicity of FCFS but also highlights its inefficiency when long processes delay shorter ones. Analysing the code first at <Line 2>, the loop for  $i = 0$  to  $n$  iterates over all processes, ensuring that each process's waiting and turnaround time is calculated. At <Line 3>, the waiting time calculation  $wt[i] = bt[i-1] + wt[i-1]$  depends on the previous process's burst and

waiting time, which is a constant-time operation, O(1). Finally, at <Line 5>, the turnaround time  $\text{tat}[i] = \text{bt}[i] + \text{wt}[i]$  is computed for each process, again O(1), resulting in an overall runtime of O(n) for n processes.

## A.2. Shortest Job First

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

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
    bool completed = false;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i=0; i<n; i++) {
        processes[i].pid = i+1;
        cout << "Enter arrival time and burst time for P" << i+1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    int completed = 0, currentTime = 0;
    float totalWT = 0, totalTAT = 0;

    while(completed != n) {
        int idx = -1;
        int minBurst = 1e9;

        for(int i=0; i<n; i++) {
            if(!processes[i].completed && processes[i].arrivalTime <= currentTime) {
                if(processes[i].burstTime < minBurst) {
                    minBurst = processes[i].burstTime;
                    idx = i;
                }
            }
        }
    }
}
```

```

if(idx != -1) {
    processes[idx].waitingTime = currentTIme - processes[idx].arrivalTime;
    processes[idx].turnAroundTime = processes[idx].waitingTime + processes[idx].burstTime;
    currentTIme += processes[idx].burstTime;
    processes[idx].completed = true;
    totalWT += processes[idx].waitingTime;
    totalTAT += processes[idx].turnAroundTime;
    completed++;
} else {
    currentTIme++;
}

cout << "\nP\tAT\tBT\tWT\tTAT\n";
for(int i=0; i<n; i++)
    cout << "P" << processes[i].pid << "\t"
        << processes[i].arrivalTime << "\t"
        << processes[i].burstTime << "\t"
        << processes[i].waitingTime << "\t"
        << processes[i].turnAroundTime << "\n";

cout << "\nAverage Waiting Time = " << totalWT/n;
cout << "\nAverage Turnaround Time = " << totalTAT/n << "\n";

return 0;
}

```

```

C:\Users\Ashley\Documents\I X + | X

Enter number of processes: 7
Enter arrival time and burst time for P1: 8
h
Enter arrival time and burst time for P2: Enter arrival time and burst time for P3: Enter arrival time and burst time
for P4: Enter arrival time and burst time for P5: Enter arrival time and burst time for P6: Enter arrival time and b
urst time for P7:
P      AT      BT      WT      TAT
P1      8      0      0      0
P2      0      0      0      0
P3      0      0      0      0
P4      0      0      0      0
P5      0      0      0      0
P6      0      0      0      0
P7      0      0      0      0

Average Waiting Time = 0
Average Turnaround Time = 0

-----
Process exited after 9.832 seconds with return value 0
Press any key to continue . . .

```

### Analysis:

The code demonstrates that processes with the shortest burst time are executed first, minimizing the overall average waiting time. The proper sorting of processes by arrival and burst time directly affects the correctness of waiting and turnaround times. The output clearly shows improved efficiency compared to FCFS, especially when short processes arrive after longer ones. At <Line 2>, the code `sort(processes.begin(), processes.end(), compareArrivalBurst)` sorts the processes by arrival and burst time. Sorting takes  $O(n \log n)$  time for  $n$  processes. Next, at <Line 5>, the loop for  $i = 0$  to  $n$  calculates waiting times by scanning previous processes to find eligible ones with minimum burst time, which in the worst case can take  $O(n^2)$  due to nested selection logic. Finally, at <Line 8>, turnaround times are computed in  $O(n)$ , giving the algorithm an overall worst-case time complexity of  $O(n^2)$  dominated by the selection step.

### B.1. Shortest Remaining Time First (SRTF)

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

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
    bool completed = false;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time and burst time for P" << i+1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }
}
```

```

int completed = 0, currentTime = 0;
float totalWT = 0, totalTAT = 0;

while(completed != n) {
    int idx = -1;
    int minBurst = 1e9;

    for(int i = 0; i < n; i++) {
        if(!processes[i].completed && processes[i].arrivalTime <= currentTime) {
            if(processes[i].burstTime < minBurst) {
                minBurst = processes[i].burstTime;
                idx = i;
            }
        }
    }

    if(idx != -1) {

        //fixed
        if(currentTime < processes[idx].arrivalTime)
            currentTime = processes[idx].arrivalTime;

        processes[idx].waitingTime = currentTime - processes[idx].arrivalTime;
        processes[idx].turnAroundTime = processes[idx].waitingTime + processes[idx].burstTime;

        currentTime += processes[idx].burstTime;
        processes[idx].completed = true;
        completed++;

        totalWT += processes[idx].waitingTime;
        totalTAT += processes[idx].turnAroundTime;

    } else {
        currentTime++;
    }
}

cout << "\nP\tAT\tBT\tWT\tTAT\n";
for(int i = 0; i < n; i++)
    cout << "P" << processes[i].pid << "\t"
        << processes[i].arrivalTime << "\t"
        << processes[i].burstTime << "\t"
        << processes[i].waitingTime << "\t"
        << processes[i].turnAroundTime << "\n";

cout << "\nAverage Waiting Time = " << totalWT / n;
cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

return 0;
}

```

```

C:\Users\Ashley\Documents\l X + | v
Enter number of processes: 8
Enter arrival time and burst time for P1: 7
9
Enter arrival time and burst time for P2: 8
0
Enter arrival time and burst time for P3: 5
6
Enter arrival time and burst time for P4: 5
7
Enter arrival time and burst time for P5: h
Enter arrival time and burst time for P6: Enter arrival time and burst time for P7: Enter arrival time and burst time
for P8:
P      AT      BT      WT      TAT
P1      7       9       11      20
P2      8       0       3       3
P3      5       6       0       6
P4      5       7       6       13
P5      0       0       0       0
P6      0       0       0       0
P7      0       0       0       0
P8      0       0       0       0

Average Waiting Time = 2.5
Average Turnaround Time = 5.25

-----
Process exited after 21.84 seconds with return value 0
Press any key to continue . . .

```

### Analysis:

The output reflects frequent preemptions, showing the CPU switching to the process with the shortest remaining burst time at each arrival. The tracking remaining times accurately is crucial for correct completion, waiting, and turnaround times. The results highlight that SRTF reduces average waiting time but requires careful handling of process arrivals and context switches. At <Line 2>, the outer loop for time = 0 to completed != n increments time unit by unit until all processes are finished, which can take up to the sum of all burst times. At <Line 4>, the inner loop scans all n processes to find the one with the smallest remaining burst time at that time unit, giving O(n) per time step. Finally, the completion and waiting times are updated in O(1) per process at <Line 6>, resulting in an overall worst-case runtime of O(n \* total\_burst\_time), which is larger than simple O(n<sup>2</sup>) algorithms for large burst sums.

### B.2. Round Robin

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Process {
    int pid;
    int burstTime;
    int remainingTime;
    int waitingTime = 0;
    int turnAroundTime = 0;
};

int main() {
    int n, quantum;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i=0; i<n; i++) {
        processes[i].pid = i+1;
        cout << "Enter burst time for P" << i+1 << ": ";
        cin >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    cout << "Enter time quantum: ";
    cin >> quantum;
    queue<int> q;
    q.push(0);
    vector<bool> inQueue(n, false);
    inQueue[0] = true;

    int currentTime = 0;
    int completed = 0;
    while(completed < n) {
        int idx = q.front(); q.pop();
```

```

int execTime = min(quantum, processes[idx].remainingTime);
processes[idx].remainingTime -= execTime;
currentTime += execTime;

for(int i=0; i<n; i++) {
    if(processes[i].remainingTime > 0 && !inQueue[i])
        q.push(i), inQueue[i] = true;
}

if(processes[idx].remainingTime > 0)
    q.push(idx);
else {
    processes[idx].turnAroundTime = currentTime;
    processes[idx].waitingTime = processes[idx].turnAroundTime - processes[idx].burstTime;
    completed++;
}

float totalWT = 0, totalTAT = 0;
cout << "\nP\tBT\tWT\tTAT\n";
for(int i=0; i<n; i++) {
    cout << "P" << processes[i].pid << "\t"
        << processes[i].burstTime << "\t"
        << processes[i].waitingTime << "\t"
        << processes[i].turnAroundTime << "\n";
    totalWT += processes[i].waitingTime;
    totalTAT += processes[i].turnAroundTime;
}

cout << "\nAverage Waiting Time = " << totalWT/n;
cout << "\nAverage Turnaround Time = " << totalTAT/n << "\n";

return 0;
}

```

```

C:\Users\Ashley\Documents\I X + ▾

Enter number of processes: 2
Enter burst time for P1: 3
Enter burst time for P2: 2
Enter time quantum: 10

P      BT      WT      TAT
P1      3       0       3
P2      2       3       5

Average Waiting Time = 1.5
Average Turnaround Time = 4

-----
Process exited after 12.66 seconds with return value 0
Press any key to continue . . .

```

## Analysis

The code output presents a cyclic execution pattern, with each process receiving fixed time slices and rejoining the queue if unfinished. These waiting times depend heavily on the time quantum and the number of cycles a process needs to finish. The output confirms fairness in scheduling, but longer processes can still have higher waiting times if the quantum is small. At <Line 2>, the code `while(!queue.empty())` executes as long as there are unfinished processes, iterating in cycles. At <Line 4>, each process is dequeued and processed for a fixed time quantum, which is  $O(1)$  per quantum. Since a process may be enqueued multiple times depending on its burst time divided by quantum, at <Line 6> the total number of iterations is  $O(\text{sum\_of\_burst\_times} / \text{quantum} * n)$ , giving the overall runtime as  $O(n * \text{total\_burst\_time} / \text{quantum})$ .

## B. Answers to Supplementary Activity

### 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.

FCFS is generally faster than non preemptive because SJF scan the ready queue repeatedly to find the shortest job because FCFS has a sort processes by arrival time  $O(n \log n)$  using a typical sorting algorithm and compute waiting time and turn around time sequentially  $O(n)$  while SJF select process with the minimum burst time from the ready queue each time a process complete and worst case selection is linear in the size of the ready queue  $O(n^2)$  in total.

FCFS has number processes 1000 with execution time 2ms while SJF with the same number of processes but execution time is 12ms making SJF slower than FCFS due to repeated search for the minimum burst time.

### 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.

RR is generally more efficient than SRTF if  $B \gg n$  as RR does not scan the entire ready queue for each time unit because SRTF checks ready queue for process with the shortest remaining burst time and total burst time of all processes is  $B$  while RR has each process slice the burst time into chunks of quantum and each time slice performs  $O(1)$  operation.

SRTF has 32 execution time units and 15 context switches while RR with the same execution time units but only 9 context switches which makes SRTF have more overhead due to repeated comparisons.

## C. Conclusion & Lessons Learned

### Summary of Lessons Learned:

#### Cruz :

I learned how different CPU scheduling algorithms affect process execution, waiting time, and turnaround time. Implementing the code helped us understand arrival times, burst times, and queue management in both non-preemptive and preemptive scheduling. I also saw how theoretical time complexity relates to practical performance.

#### Escalona:

I gained practical experience in coding CPU scheduling algorithms and analyzing their efficiency using theoretical and empirical methods. I learned how metrics like waiting time, turnaround time, and response time differ across scheduling strategies, emphasizing the importance of proper algorithm selection. Overall, the activity reinforced the connection between theoretical knowledge and practical implementation, preparing me for more advanced tasks in operating system management.

#### Dano:

The activity explored both non-preemptive and preemptive CPU scheduling algorithms, focusing on their computational behavior and runtime efficiency. By applying theoretical runtime analysis and step-by-step code examination, I observed how algorithms like FCFS, SJF, SRTF, and Round Robin handle process selection, waiting, and turnaround times. The study highlighted the contrast between simple, low-overhead algorithms and more complex, resource-intensive approaches, providing a clear understanding of trade-offs in efficiency and fairness.

#### Mondragon:

The activity explored non-preemptive and preemptive CPU scheduling algorithms, examining how FCFS, SJF, SRTF, and Round Robin handle process selection, waiting, and turnaround times. The study highlighted differences in efficiency and fairness, providing insights into trade-offs between computational cost and scheduling performance.

### **Analysis of the Procedure:**

#### **Cruz:**

Following the step-by-step implementation clarified how FCFS, SJF, SRTF, and Round Robin select processes. Debugging the code taught me the importance of careful array and loop management for accurate calculations. Overall, the procedure strengthened our logical thinking in algorithm design.

#### **Escalona:**

In my analysis, it revealed to me how different scheduling algorithms vary in computational cost, with some requiring sorting while others depend on constant preemption decisions, making the contrast between efficient and more resource-heavy schedulers clearer. In the same way, examining the code showed that at Line 2, the loop for (`int i = 0; i < y.length; i++`) ensures every element of `y` is processed exactly `n` times, and at Line 3, `search(x, y[i])` may scan all `m` elements of `x` during each iteration. Debugging both the scheduling procedures and this search algorithm emphasized the importance of precise array and loop management to ensure correct and efficient behavior.

#### **Dano:**

I used theoretical runtime analysis to understand how the algorithm behaves as the input grows. At Line 2, the loop for(`int i = 0; i < y.length; i++`) ensures every element of array `y` is examined, running exactly `n` times in the worst case. At Line 3, the condition `if(search(x, y[i]) != -1)` calls the `search()` function for each element of `y`, which may examine all `m` elements of `x`, performing up to `m` comparisons per iteration. Finally, if no common value is found, the loop completes all `n` iterations, and Line 7 executes `return true;`, completing the full comparison process. Step-by-step debugging of this routine highlighted the importance of proper array and loop management to ensure accurate behavior.

#### **Mondragon:**

Implementing each scheduling algorithm in C++ allowed me to observe their practical execution and measure key metrics like waiting time and turnaround time. The step-by-step procedure, including Gantt chart preparation and average waiting time computation, highlighted how theoretical concepts translate into real-world performance. This hands-on approach improved my understanding of algorithm behavior and coding implementation challenges.

### **Analysis of the Supplementary Activity:**

#### **Cruz:**

Simulating processes and calculating waiting and turnaround times helped us verify algorithm correctness. Creating Gantt charts gave practical insight into CPU utilization. This hands-on activity enhanced my understanding beyond theory.

#### **Escalona:**

I realized that the supplementary activity showed differences in CPU scheduling algorithms in terms of computational cost and efficiency. Non-preemptive algorithms like FCFS are faster due to sequential processing, while SJF and preemptive algorithms like SRTF require repeated scanning and more comparisons, increasing

overhead, whereas Round Robin executes fixed time slices with fewer operations. Comparing these behaviors alongside the search routine clarified the contrast between simple, low-overhead algorithms and more resource-intensive ones.

**Dano:**

The supplementary activity showed how CPU scheduling algorithms differ in computational cost and efficiency. Non-preemptive algorithms like FCFS are faster due to sequential processing ( $O(n)$  after sorting), while SJF requires repeated scanning for the shortest job, leading to  $O(n^2)$  in the worst case. Among preemptive algorithms, SRTF frequently checks the ready queue to find the process with the shortest remaining burst time, resulting in more comparisons and context switches, whereas Round Robin executes fixed time slices with fewer overhead operations. Comparing these behaviors alongside the search routine clarified the differences between simple, low-overhead algorithms and more resource-intensive approaches, reinforcing the importance of careful implementation and stepwise analysis.

**Mondragon:**

The supplementary activity illustrated how CPU scheduling algorithms vary in efficiency and computational demands. FCFS runs quickly with its straightforward sequential processing, while SJF and SRTF incur extra overhead from repeated checks and comparisons, and Round Robin balances workload with fixed time slices. Observing these alongside the search routine highlighted the gap between lightweight, efficient methods and those that require more resources.

**Concluding Statement:**

**Cruz:**

I think I performed well in implementing and testing the algorithms. My main area for improvement is optimizing code for larger datasets. Overall, this activity deepened my understanding of CPU scheduling.

**Escalona:**

Through this activity, I learned the strengths and limitations of both preemptive and non-preemptive scheduling algorithms and how they impact CPU performance. I realized that while I can implement basic algorithms effectively, I need to improve on optimizing more complex preemptive algorithms like SRTF and understanding their overhead in dynamic environments. Developing a deeper insight into time complexity analysis and efficient coding practices will enhance my proficiency in scheduling algorithms.

**Dano:**

The activity strengthened my understanding of algorithm efficiency, runtime complexity, and the trade-offs between fairness and computational cost. Careful debugging and step-by-step analysis improved my logical thinking and prepared me to design algorithms that balance correctness and performance.

**Mondragon:**

In conclusion, I learned how different CPU scheduling algorithms vary in efficiency, computational cost, and overhead, and how their behaviors affect overall performance. I still need to deepen my understanding of optimizing more complex algorithms like SJF and SRTF and improve my ability to implement them efficiently in practical scenarios

<b>E. External References</b>