Simulation of A Core Algorithm of Operating System

By: Kimia Esmaili (Stu. ID: 610398193)
Professor: Ali Reza Khalilian

February 2022

Table of contents:

Abstract:

The objective of this project is to simulate one of the core algorithms of operating system in C++ programming language. This simulation has a statistical approach where we analyze the result and see how changing different parameters would affect the result. The chosen algorithm for this study is FCFS (First Come, First Serve) algorithm which is a CPU scheduling algorithm. First, we will review some general concepts and requirements for our work. We will also compare CPU scheduling algorithms and see some pros and cons of each and mention which algorithm is more suitable for a certain process.

Introduction:

Task scheduling is needed to maintain every process that comes with a processor in parallel processing. In several conditions, not every algorithm works better on the significant problem. However, it cannot be predicted what process will come after. Average Waiting Time is a standard measure for giving credit to the scheduling algorithm. Several techniques have been applied to maintain the process to make the CPU performance in normal.

Scheduling is already part of a parallel process. In scheduling, there are several methods used to perform queue process that comes to the processor. Some algorithms are popular among other First Come First Serve, Shortest Job First, and Round Robin. In this study, we also see the average waiting time of each of FCFS algorithm. The priority of the processes that occur on the processor is something that determines when the process will be done. However, it does not discuss the priority in this study. The scheduling assumes all of which occurred in the queue have the same priority value (but we use FCFS). We will further explore the concepts of scheduling.

CPU Scheduling:

A process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution. The process is the state when a program is executed. When the computer is running, there are many processes running simultaneously. A process may create a derivative process is carried out by a parent process. The derivation process is also able to create a new process so that all of these processes ultimately forming process tree. When a process is made then the process can obtain these resources such as CPU time, memory, files, or I/O devices. These resources can be obtained directly from the operating system, from the parent process that dispenses resources to each process its derivative, or the derivative and the parent process share the resources of a given operating system. CPU scheduling is part of a multi-programming operating system.

Basic concepts:

Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)

In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.

A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.

The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

Types of scheduling and how to recognize them:

Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling. FCFS is a non-preemptive algorithm.

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

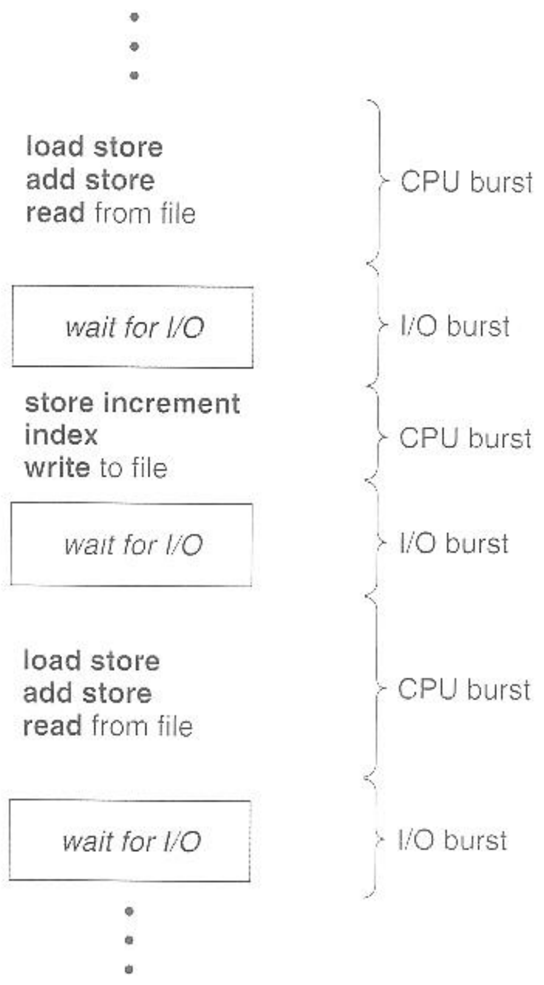
1. A process switches from the running to the waiting state.
2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called non- preemptive. All other scheduling are preemptive.

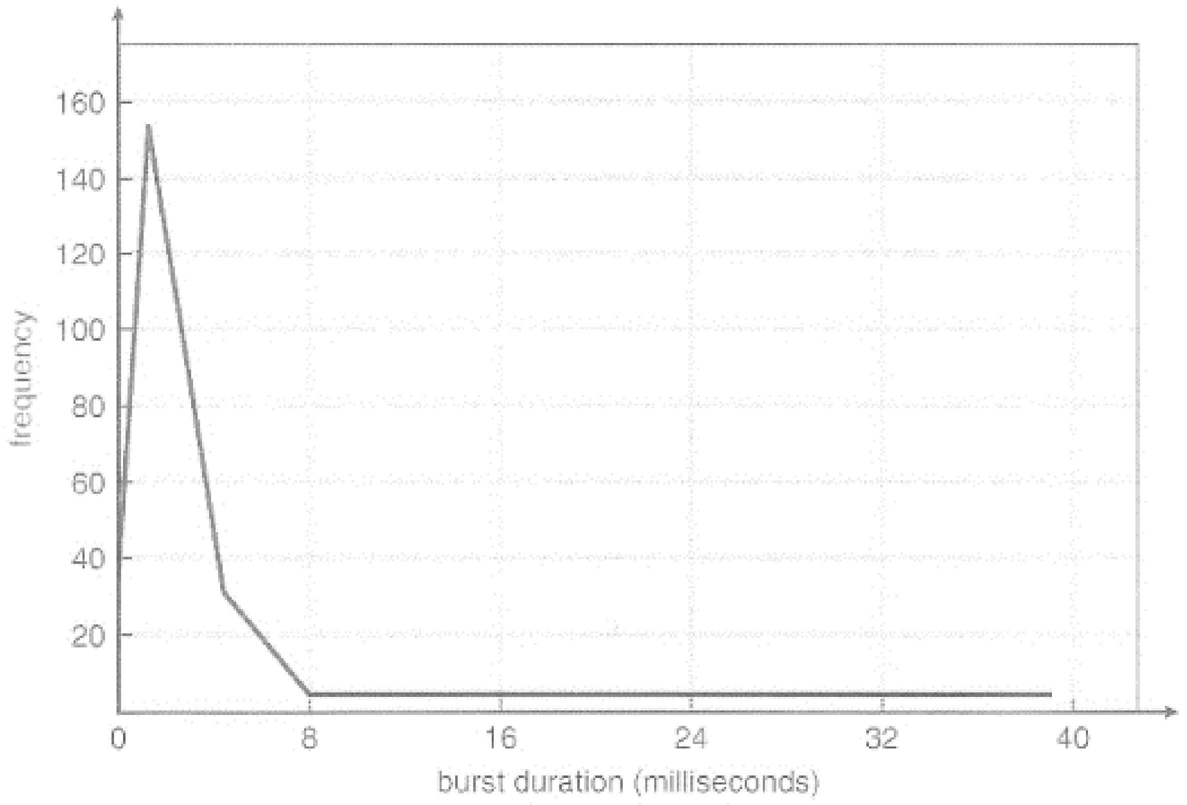
CPU-I/O Burst Cycle:

Almost all processes alternate between two states in a continuing ***cycle***, as shown in the figure below***:***

- A CPU burst of performing calculations, and
- An I/O burst, waiting for data transfer in or out of the system.



CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in the figure below:



Important CPU scheduling Terminologies:

When we are dealing with some CPU scheduling algorithms then we encounter with some confusing terms like Burst time, Arrival time, Exit time, Waiting time, Response time, Turnaround time, and throughput. These parameters are used to find the performance of a system. So, in this blog, we will learn about these parameters. Let's get started one by one.

Burst time:

Every process in a computer system requires some amount of time for its execution. This time is both the CPU time and the I/O time. The CPU time is the time taken by CPU to execute the process. While the I/O time is the time taken by the process to perform some I/O operation. In general, we ignore the I/O time and we consider only the CPU time for a process. So, Burst time is the total time taken by the process for its execution on the CPU.

Arrival time:

Arrival time is the time when a process enters into the ready state and is ready for its execution.

Process	Arrival time	Burst time
P1	0 ms	8 ms
P2	1 ms	7 ms
P3	2 ms	10 ms

Here in the above example, the arrival time of all the 3 processes are 0 ms, 1 ms, and 2 ms respectively.

Exit time:

Exit time is the time when a process completes its execution and exit from the system.

Response time:

Response time is the time spent when the process is in the ready state and gets the CPU for the first time. For example, here we are using the First Come First Serve CPU scheduling algorithm for the below 3 processes:

Process	Arrival time	Burst time
P1	0 ms	8 ms
P2	1 ms	7 ms
P3	2 ms	10 ms

Here, the response time of all the 3 processes are:

- P1: 0 ms
- P2: 7 ms because the process P2 have to wait for 8 ms during the execution of P1 and then after it will get the CPU for the first time. Also, the arrival time of P2 is 1 ms. So, the response time will be 8-1 = 7 ms.
- P3: 13 ms because the process P3 have to wait for the execution of P1 and P2 i.e. after 8+7 = 15 ms, the CPU will be allocated to the process P3 for the first time. Also, the arrival of P3 is 2 ms. So, the response time for P3 will be 15-2 = 13 ms.

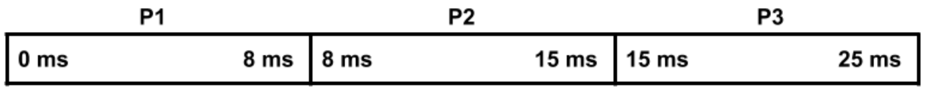
Response time =
Time at which the process gets the CPU for the first time - Arrival time

Waiting time:

Waiting time is the total time spent by the process in the ready state waiting for CPU. For example, consider the arrival time of all the below 3 processes to be 0 ms, 0 ms, and 2 ms and we are using the First Come First Serve scheduling algorithm.

Process	Arrival time	Burst time
P1	0 ms	8 ms
P2	0 ms	7 ms
P3	2 ms	10 ms

Gantt Chart



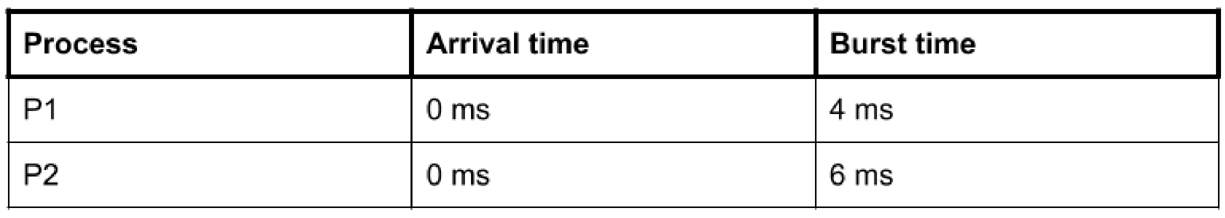
Then the waiting time for all the 3 processes will be:

- **P1:** 0 ms
- **P2:** 8 ms because P2 have to wait for the complete execution of P1 and arrival time of P2 is 0 ms.
- **P3:** 13 ms becuase P3 will be executed after P1 and P2 i.e. after 8+7 = 15 ms and the arrival time of P3 is 2 ms. So, the waiting time of P3 will be: 15-2 = 13 ms.

Waiting time = Turnaround time - Burst time

In the above example, the processes have to wait only once. But in many other scheduling algorithms, the CPU may be allocated to the process for some time and then the process will be moved to the waiting state and again after some time, the process will get the CPU and so on.

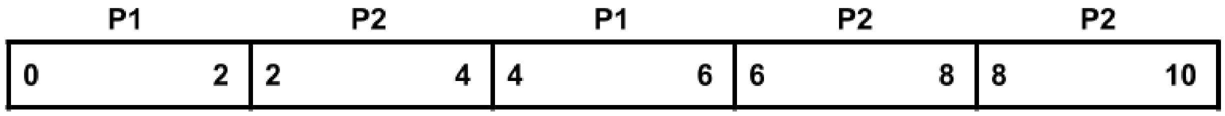
There is a difference between waiting time and response time. Response time is the time spent between the ready state and getting the CPU for the first time. But the waiting time is the total time taken by the process in the ready state. Let's take an example of a round-robin scheduling algorithm. The time quantum is 2 ms.



Process	Arrival time	Burst time
P1	0 ms	4 ms
P2	0 ms	6 ms

Time Quantum = 2ms

Gantt Chart



In the above example, the response time of the process P2 is 2 ms because after 2 ms, the CPU is allocated to P2 and the waiting time of the process P2 is 4 ms i.e turnaround time - burst time (10 - 6 = 4 ms).

Turnaround time:

Turnaround time is the total amount of time spent by the process from coming in the ready state for the first time to its completion.

Turnaround time = Burst time + Waiting time

or

Turnaround time = Exit time - Arrival time

For example, if we take the First Come First Serve scheduling algorithm, and the order of arrival of processes is P1, P2, P3 and each process is taking 2, 5, 10 seconds. Then the turnaround time of P1 is 2 seconds because when it comes at 0th second, then the CPU is allocated to it and so the waiting time of P1 is 0 sec and the turnaround time will be the Burst time only i.e. 2 seconds. The turnaround time of P2 is 7 seconds because the process P2 have to wait for 2 seconds for the execution of P1 and hence the waiting time of P2 will be 2 seconds. After 2 seconds, the CPU will be given to P2 and P2 will execute its task. So, the turnaround time will be 2+5 = 7 seconds. Similarly, the turnaround time for P3 will be 17 seconds because the waiting time of P3 is 2+5 = 7 seconds and the burst time of P3 is 10 seconds. So, turnaround time of P3 is 7+10 = 17 seconds.

Different CPU scheduling algorithms produce different turnaround time for the same set of processes. This is because the waiting time of processes differ when we change the CPU scheduling algorithm.

Throughput:

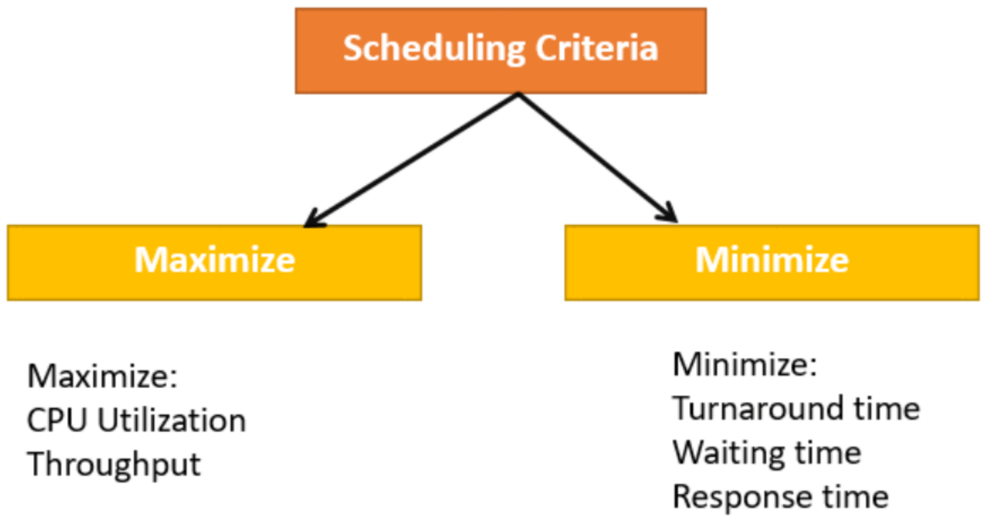
Throughput is a way to find the efficiency of a CPU. It can be defined as the number of processes executed by the CPU in a given amount of time. For example, let's say, the process P1 takes 3 seconds for execution, P2 takes 5 seconds, and P3 takes 10 seconds. So, throughput, in this case, the throughput will be (3+5+10)/3 = 18/3 = 6 seconds.

CPU utilization:

CPU utilization is the main task in which the operating system needs to make sure that CPU remains as busy as possible. It can range from 0 to 100 percent. However, for the RTOS, it can be range from 40 percent for low-level and 90 percent for the high-level system.

CPU Scheduling Criteria:

A CPU scheduling algorithm tries to maximize and minimize the following:



Interval Timer:

Timer interruption is a method that is closely related to preemption. When a certain process gets the CPU allocation, a timer may be set to a specified interval. Both timer interruption and preemption force a process to return the CPU before its CPU burst is complete.

Most of the multi-programmed operating system uses some form of a timer to prevent a process from tying up the system forever.

Dispatcher:

It is a module that provides control of the CPU to the process. The Dispatcher should be fast so that it can run on every context switch. Dispatch latency is the amount of time needed by the CPU scheduler to stop one process and start another.

Functions performed by Dispatcher:

- Context Switching
- Switching to user mode
- Moving to the correct location in the newly loaded program.

“First Come, First Serve” algorithm:

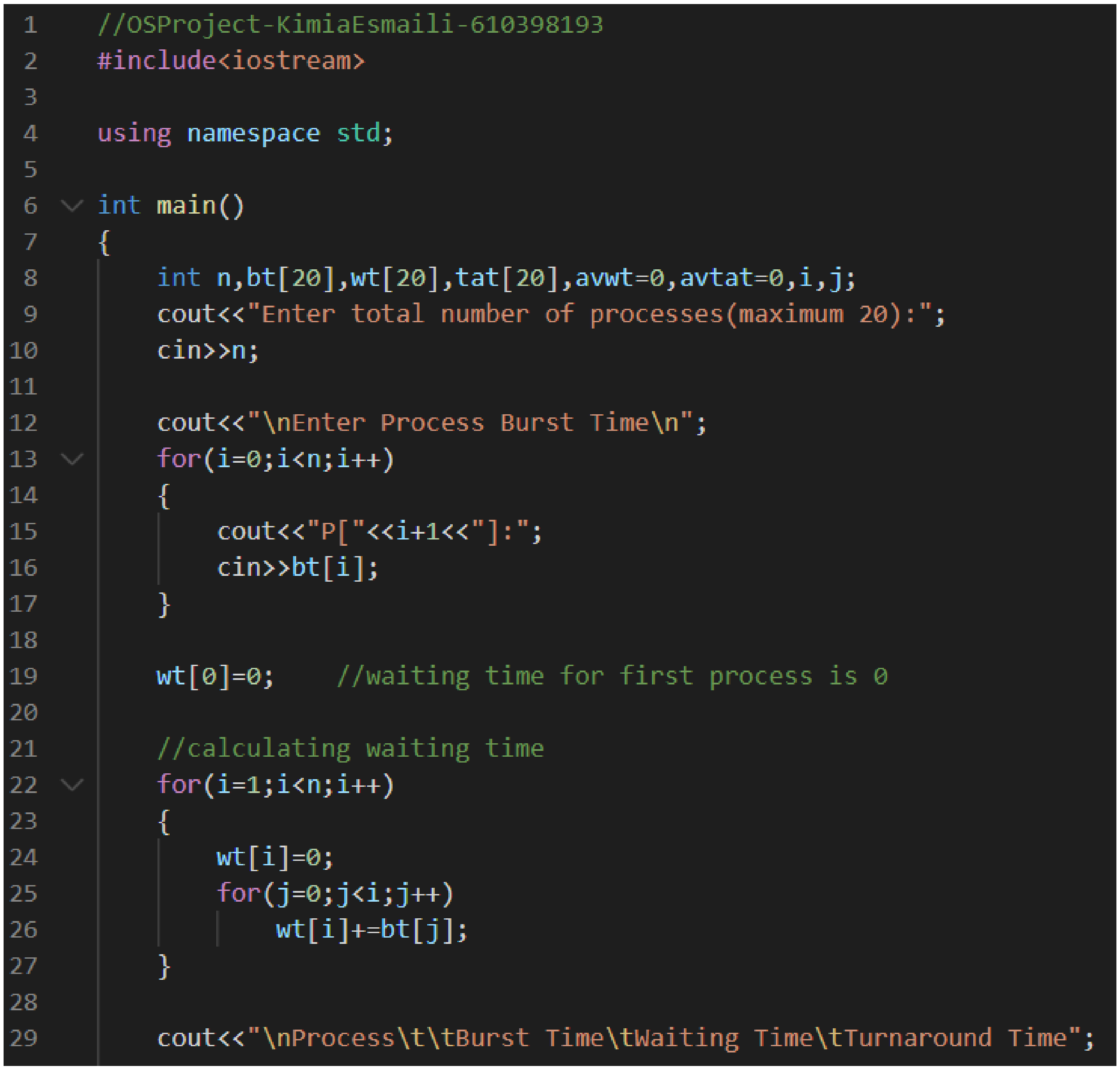
First Come First Serve is the full form of FCFS. It is the easiest and most simple CPU scheduling algorithm (FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine). In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

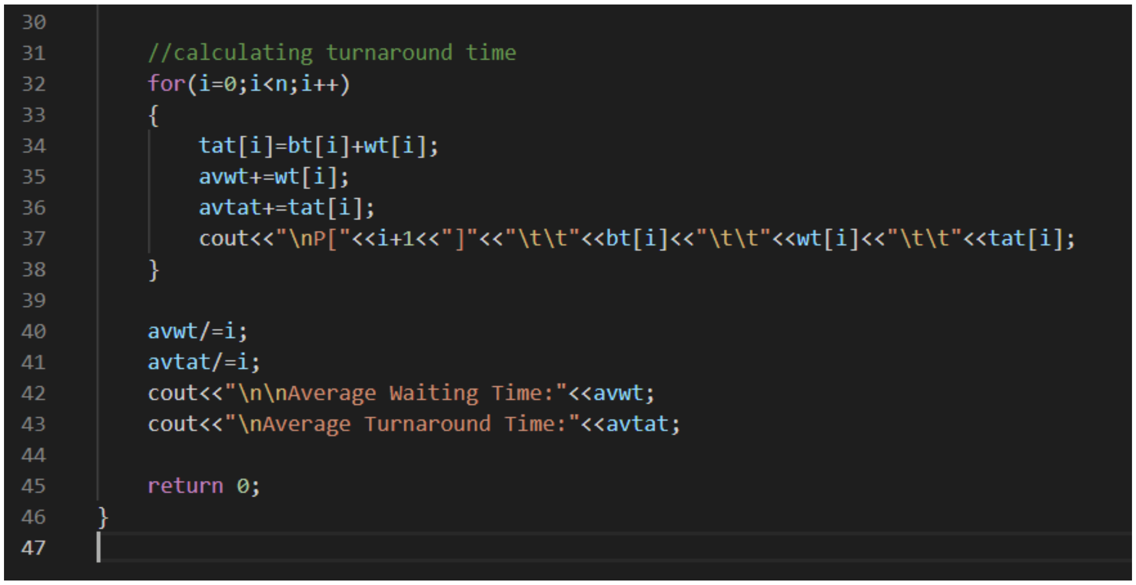
Characteristics of FCFS method:

- It offers non-preemptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis
- It is easy to implement and use.
- However, this method is poor in performance, and the general wait time is quite high.

Code implementation of FCFS:

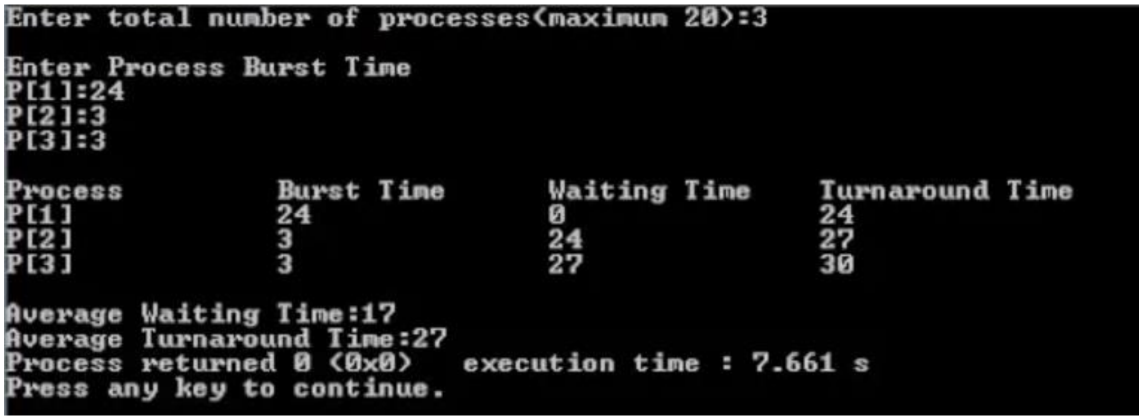


```
//OSProject-KimiaEsmaili-610398193
#include<iostream>

using namespace std;

int main()
{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    cout<<"Enter total number of processes(maximum 20):";
    cin>>n;

    cout<<"\nEnter Process Burst Time\n";
    for(i=0;i<n;i++)
    {
        cout<<"P["<<i+1<<"]:";
        cin>>bt[i];
    }

    wt[0]=0;    //waiting time for first process is 0

    //calculating waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }

    cout<<"\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time";
```



```
    //calculating turnaround time
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        cout<<"\nP["<<i+1<<"]"<<"\t\t"<<bt[i]<<"\t\t"<<wt[i]<<"\t\t"<<tat[i];
    }

    avwt/=i;
    avtat/=i;
    cout<<"\n\nAverage Waiting Time:"<<avwt;
    cout<<"\nAverage Turnaround Time:"<<avtat;

    return 0;
}
```

Output:



```
Enter total number of processes(maximum 20):3

Enter Process Burst Time
P[1]:24
P[2]:3
P[3]:3

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            24              0               24
P[2]            3               24              27
P[3]            3               27              30

Average Waiting Time:17
Average Turnaround Time:27
Process returned 0 (0x0)   execution time : 7.661 s
Press any key to continue.
```

Explenation:

1- Input the processes along with their burst time (bt).

2- Find waiting time (wt) for all processes.

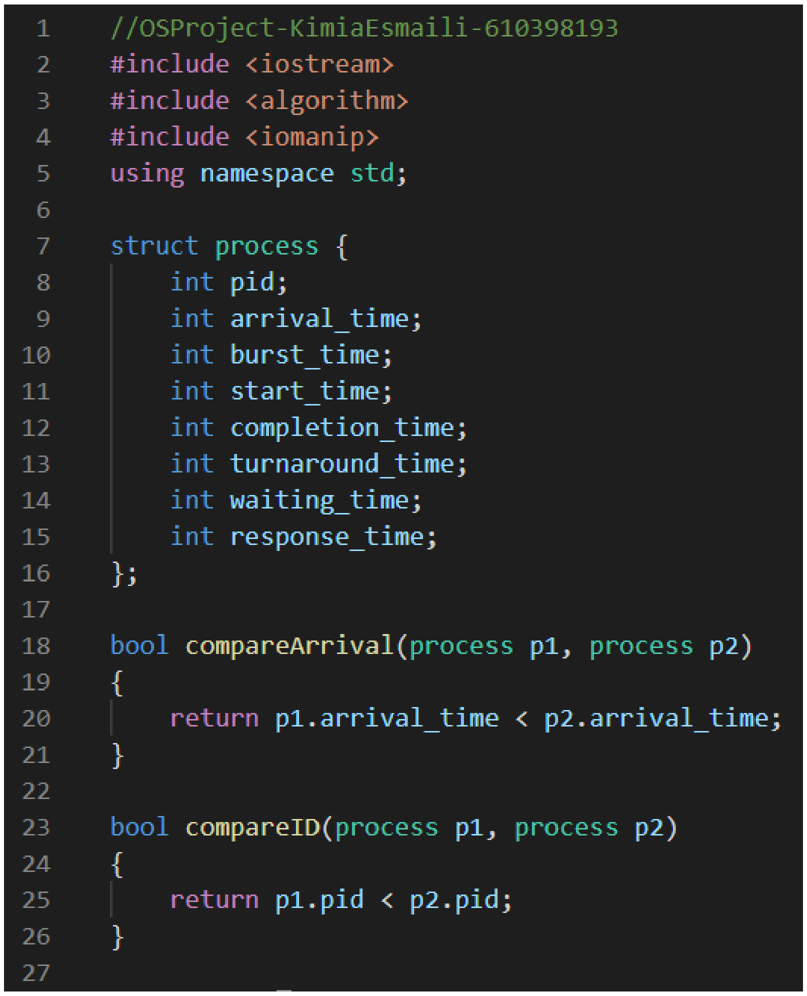
3- As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e. wt[0] = 0.

4- Find **waiting time** for all other processes i.e. for
process i ->
wt[i] = bt[i-1] + wt[i-1] .

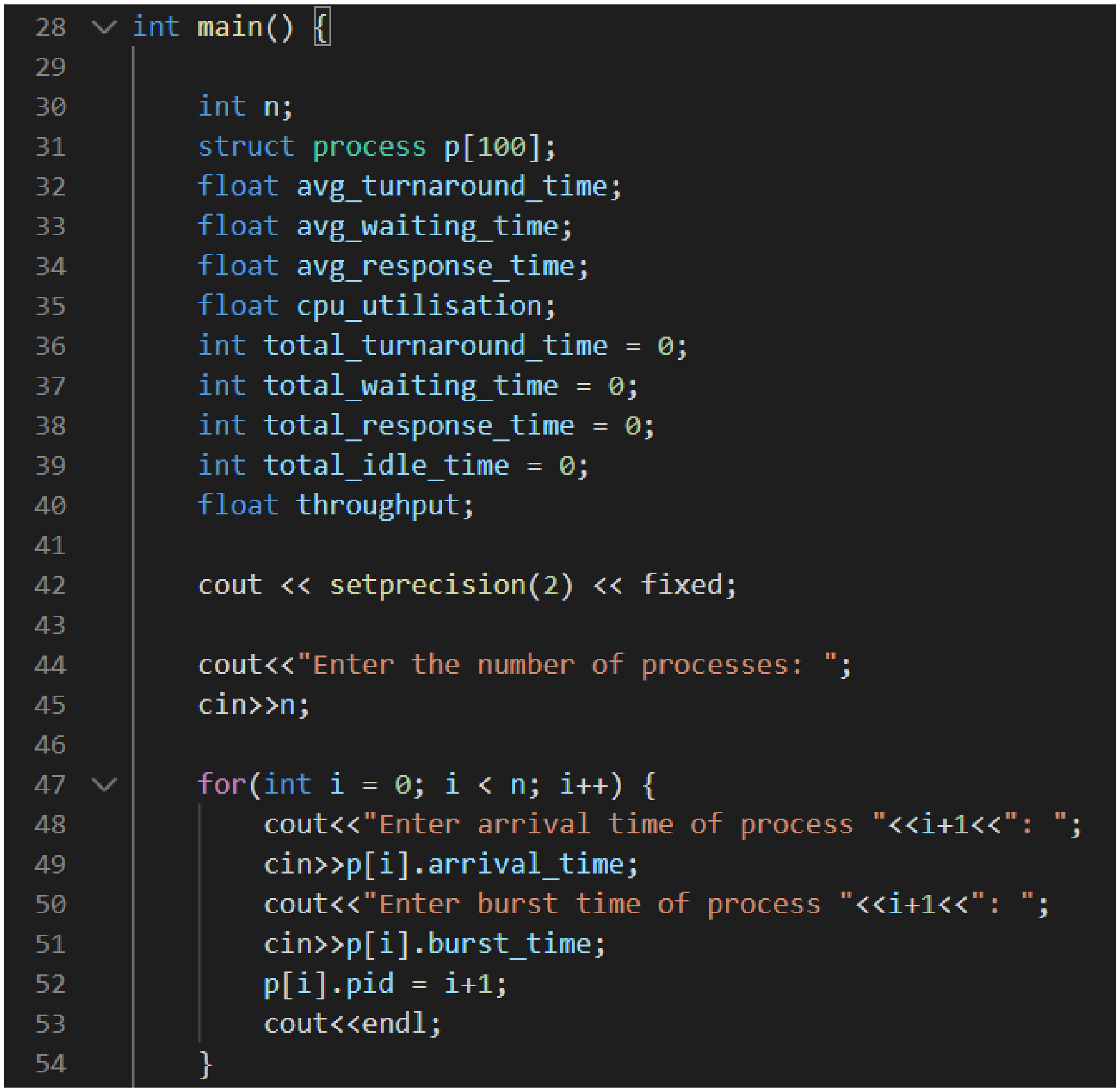
5- Find **turnaround time** = waiting_time + burst_time
for all processes.

6- Find **average waiting time** =
total_waiting_time / no_of_processes.
7- Similarly, find **average turnaround time** =
total_turn_around_time / no_of_processes.

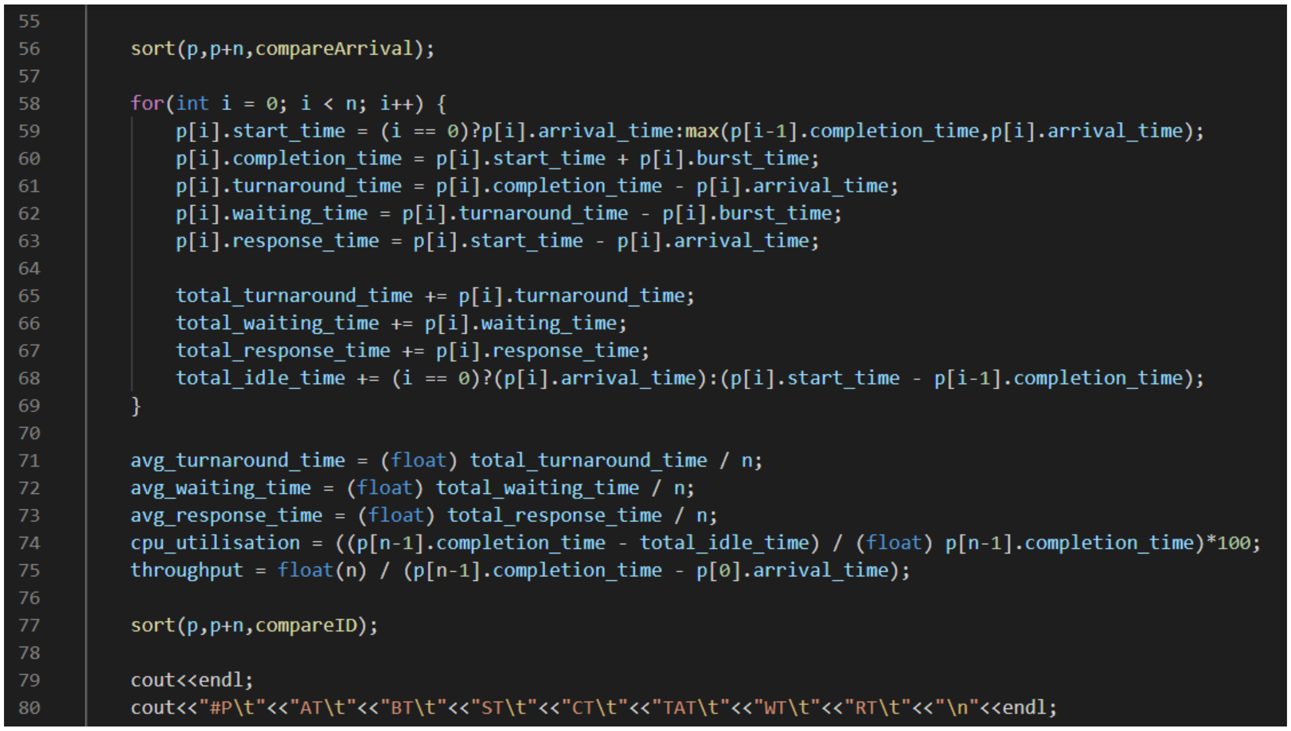
And another implementation for some parameters:



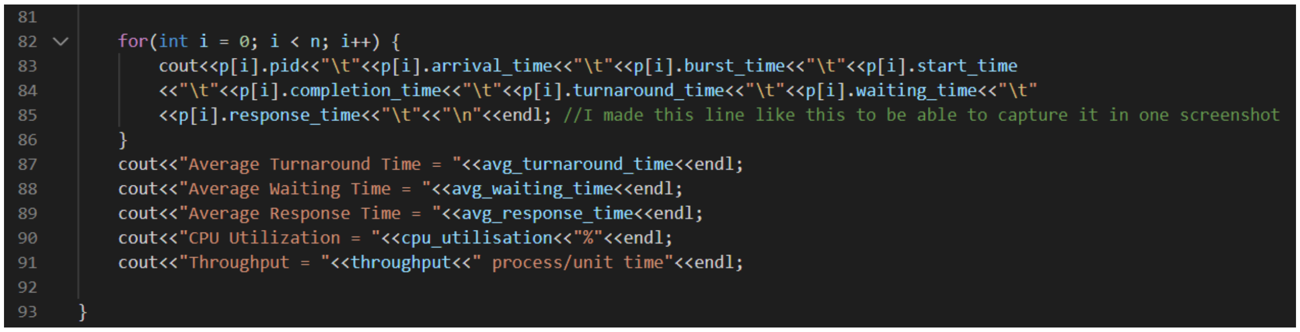
```
//OSProject-KimiaEsmaili-610398193
#include <iostream>
#include <algorithm>
#include <iomanip>
using namespace std;

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};

bool compareArrival(process p1, process p2)
{
    return p1.arrival_time < p2.arrival_time;
}

bool compareID(process p1, process p2)
{
    return p1.pid < p2.pid;
}

```



```cpp
int main() {

    int n;
    struct process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
    float avg_response_time;
    float cpu_utilisation;
    int total_turnaround_time = 0;
    int total_waiting_time = 0;
    int total_response_time = 0;
    int total_idle_time = 0;
    float throughput;

    cout << setprecision(2) << fixed;

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

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



```cpp

    sort(p,p+n,compareArrival);

    for(int i = 0; i < n; i++) {
        p[i].start_time = (i == 0)?p[i].arrival_time:max(p[i-1].completion_time,p[i].arrival_time);
        p[i].completion_time = p[i].start_time + p[i].burst_time;
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
        p[i].response_time = p[i].start_time - p[i].arrival_time;

        total_turnaround_time += p[i].turnaround_time;
        total_waiting_time += p[i].waiting_time;
        total_response_time += p[i].response_time;
        total_idle_time += (i == 0)?(p[i].arrival_time):(p[i].start_time - p[i-1].completion_time);
    }

    avg_turnaround_time = (float) total_turnaround_time / n;
    avg_waiting_time = (float) total_waiting_time / n;
    avg_response_time = (float) total_response_time / n;
    cpu_utilisation = ((p[n-1].completion_time - total_idle_time) / (float) p[n-1].completion_time)*100;
    throughput = float(n) / (p[n-1].completion_time - p[0].arrival_time);

    sort(p,p+n,compareID);

    cout<<endl;
    cout<<"#P\t"<<"AT\t"<<"BT\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;
```



```cpp

    for(int i = 0; i < n; i++) {
        cout<<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"<<"\n"<<endl; //I made this line like this to be able to capture it in one screenshot
    }
    cout<<"Average Turnaround Time = "<<avg_turnaround_time<<endl;
    cout<<"Average Waiting Time = "<<avg_waiting_time<<endl;
    cout<<"Average Response Time = "<<avg_response_time<<endl;
    cout<<"CPU Utilization = "<<cpu_utilisation<<"%"<<endl;
    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;

}
```

Code explanation:

Input:

1. Number of processes
2. Arrival time of each process. If all process arrives at the same time, this can be set to 0 for all processes.
3. Burst time of each process

Output:

1. Start Time (ST), Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT) and Response Time (RT) for each process.
2. Average turnaround Time, average waiting time and average response time.
3. Throughput and CPU utilization.

Algorithm:

1. Stable sort the processes in order of arrival time in ascending order.
2. Calculate start time and completion time for each process
3. Start time = (i == 0)?p[i].arrival_time:max(p[i].arrival_time,p[i-1].completion_time)
4. completion_time = start_time + p[i].burst_time
5. Calculate TAT, WT and RT using formulas:

   TAT = CT - AT

   WT = TAT - BT

   RT = ST – AT

As we said, FCFS can be defined as a process that arrives first will be served first. If there is a process to arrive at the same time, the
services they carried through their order in the queue. The process in the queue behind had to wait until all the process in front of him is complete. Any process that is on ready status put in FCFS queue according to the time of arrival. The process which has been done does enter the queue anymore. However, here the process which burst time is short must wait for the other process to be finished.

Analysis:

The creation of data sampling process must be established. Some of them have the same arrival time and different burst time.
The table below shows the data in the process:
Data sampling.

Process	Arrival Time	Burst Time
P1	0	1
P2	0	1
P3	0	1
P4	3	1
P5	3	2
P6	3	3
P7	7	3
P8	7	2
P9	7	1
P10	13	1
P11	13	2
P12	13	3
P13	17	5
P14	17	2
P15	19	4
P16	21	7
P17	23	3
P18	24	5
P19	24	1
P20	24	3
P21	24	5
P22	24	4
P23	25	8
P24	25	4
P25	26	3

Let’s consider that Process is P, Arrival Time is AT, Burst Time is BT, Waiting Time is WT. The formula below is to obtain the waiting time.

1- $\boldsymbol{WTn} = (\Sigma\, \boldsymbol{BT}) - \boldsymbol{ATn}$
Having $\boldsymbol{n-1}$,$\boldsymbol{i=1}$

2- $\boldsymbol{TWT} = \Sigma\, \boldsymbol{WT}$

Every single waiting time must be calculated. Then calculate the Total Waiting Time and the Average Waiting Time of the FCFS
algorithm.
WT1 = P1AT
= 0
WT2 = P1BT - P2AT
= 1 - 0
= 1
WT3 = P1BT + P2BT - P3AT
= 1 + 1 - 0
= 2
WT4 = P1BT + P2BT + P3BT – P4AT
= 1 + 1 + 1 - 3
= 0
WT5 = P1BT + P2BT + P3BT + P4BT – P5AT
= 1 + 1 + 1 + 1 - 3
= 1
WT25 = P1BT + P2BT + P3BT + … + P24BT – P25AT
= 1 + 1 + 1 +1 + ... + 4 – 26
= 46

The calculation continues until the last process. It produces the total waiting time and finally the average waiting time is obtained.

TWT = WT1 + WT2 + WT3 + WT4 + WT5 + ... + WT25
= 0 + 1 + 2 + 0 + 1 + ... + 46
= 328
AWT = TWT / TOTAL PROCESS
= 328 / 25
= **13.12**

Average Waiting Time is a standard measure for giving credit to the scheduling algorithm. The standard deviation was also used. The population standard deviation formula is given as:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (X_i - \mu)^2}$$

Here,
σ = Population standard deviation
μ = Assumed mean

Similarly, the sample standard deviation formula is:

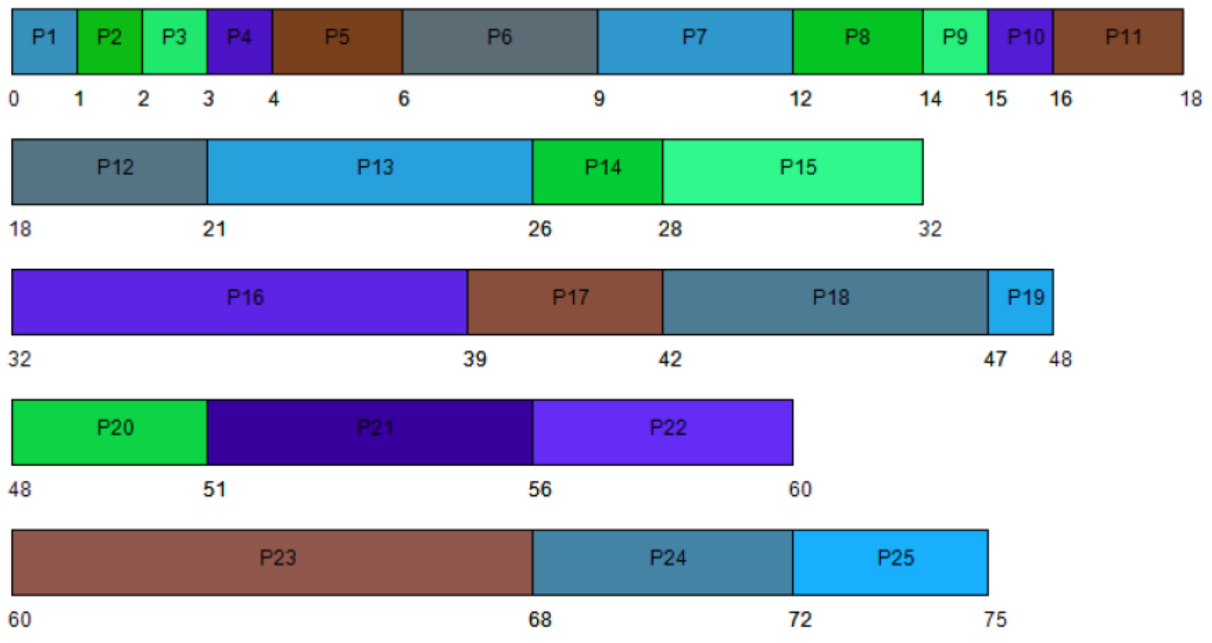
$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Here,
s = Sample standard deviation
¯xx¯ = Arithmetic mean of the observations

So changing any parameter can distribute to change in the results, but the efficiency of the algorithm is the same.

Imagine the incoming processes are short and there is no need for the processes to execute in a specific order. In this case, FCFS works best when compared to other algorithms (such as SJF and RR) because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

The figure below shows the time split which represents each process of the FCFS algorithm:



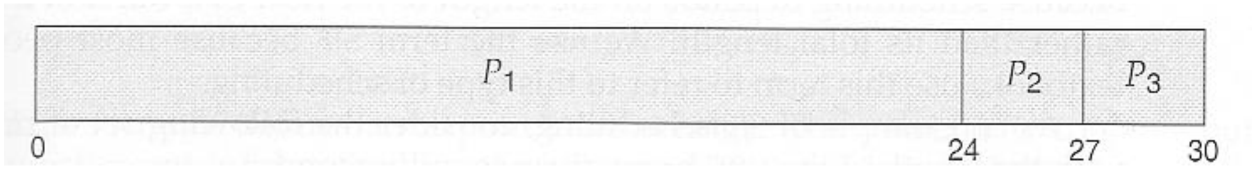
Result:

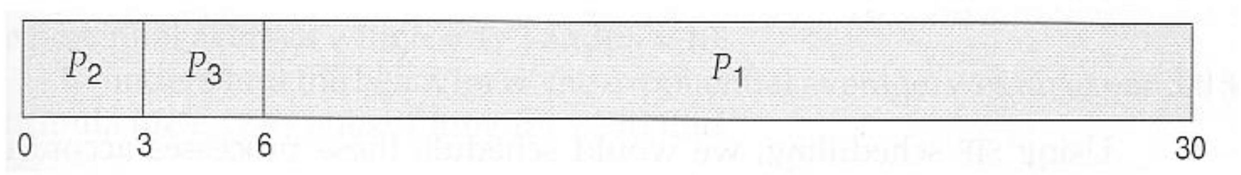
As we can see, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

Process	Burst Time
P1	24
P2	3
P3	3

In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.

In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.





FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect*** (we will discuss this concept later). When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Convoy Effect:

A phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.

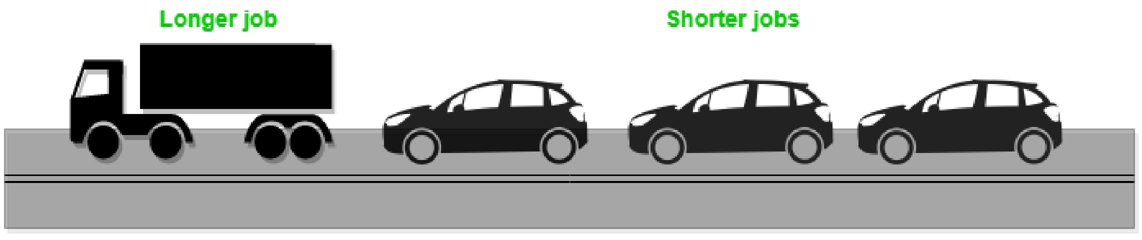


Figure - The Convey Effect, Visualized

FCFS algorithm is non-preemptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation called Convoy Effect.

Suppose there is one CPU intensive (large burst time) process in the ready queue, and several other processes with relatively less burst times but are Input/Output (I/O) bound (Need I/O operations frequently).

Steps are as following below:

- The I/O bound processes are first allocated CPU time. As they are less CPU intensive, they quickly get executed and goto I/O queues.
- Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
- While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
- However, the I/O bound processes are made to wait as the CPU intensive process still hasn't finished. **This leads to I/O devices being idle.**
- When the CPU intensive process gets over, it is sent to the I/O queue so that it can access an I/O device.

- Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
- However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, **the CPU is sitting idle now**.

Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.

To avoid Convoy Effect, preemptive scheduling algorithms like Round Robin Scheduling can be used – as the smaller processes don’t have to wait much for CPU time – making their execution faster and leading to less resources sitting idle.

A Brief Comparison:

We know that how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of some scheduling algorithms and some examples of which algorithm is best for a situation (Sometimes FCFS algorithm is better than the other in short burst time while Round Robin is better for multiple processes in every single time.).

First Come First Serve (FCFS):

Let's start with the **Advantages:**

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

It's time for the **Disadvantages:**

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.

- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

Shortest Job First (SJF):

Starting with the **Advantages:** of Shortest Job First scheduling algorithm.

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

And the **Disadvantages:**

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Note: Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

Round Robin (RR):

Here are some **Advantages:** of using the Round Robin Scheduling:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

and here comes the **Disadvantages:**

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
- If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

Priority based Scheduling:

Advantages of Priority Scheduling:

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Some **Disadvantages:**

- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Usage of Scheduling Algorithms in Different Situations:

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

Situation 1: (mentioned before)

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

Situation 2:

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

Situation 3:

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.

Reference:

A. Silberschatz, P. Galvin, G. Gagne, “Operating Systems Concepts (8th Edition)”, Wiley Pvt. Ltd.