***Operating System***

***Final project***

***(CPU Scheduling Algorithms)***

|  |  |  |
| --- | --- | --- |
| **Team**  **No.2** | **Name** | **ID** |
| **1** | **Mariam El-Habashi** | 22010394 |
| **2** | **Salma Soliman** | 22010111 |
| **3** | **Rawan Ibrahim** | 22010092 |
| **4** | **Rana Samir** | 22011607 |
| **5** | **Shahd Waleed** | 22010119 |
| **6** | **Hamass Nagieb Mortada** | 22010331 |
| **7** | **Shaimaa Mohamed Ahmed** | 22010120 |

*1st FCFS Scheduling:*

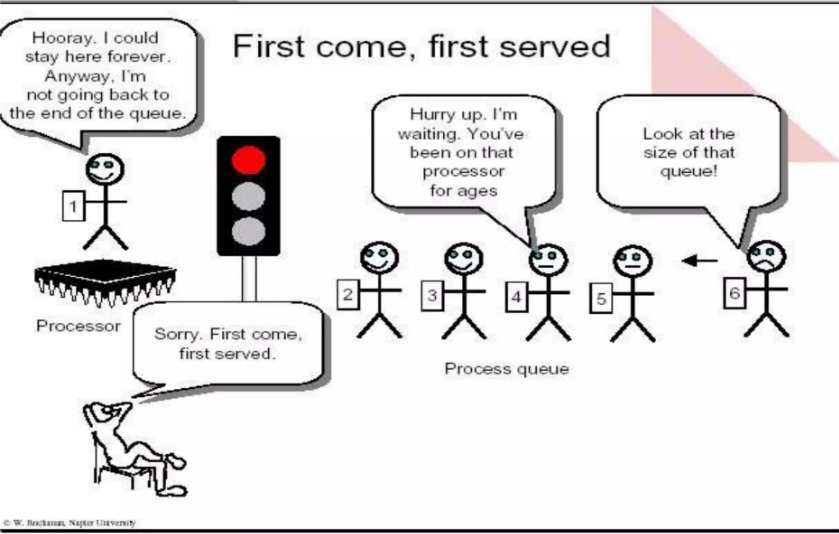
**First Come First Serve (FCFS)** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival.

It is the Easiest and Simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.

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

**Characteristics of FCFS method:**

* It supports 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.
* This method is poor in performance, and the general wait time is quite high



**FCFS Implementation and Results:**

**Current Time:** The **current time** tracks when the CPU is available for the next process and is updated as each process completes. It starts at **0**, and after each process finishes, the **current time** advances by the duration of the process's burst time.

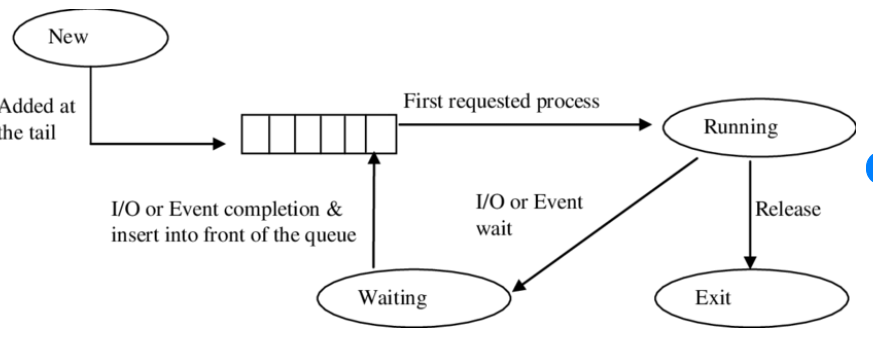
For example:

* **P4** starts execution at time 0 and has a burst time of 3, the **current time** will be 3 after **P4** completes.

1. **Execution Order (Gantt Chart):**

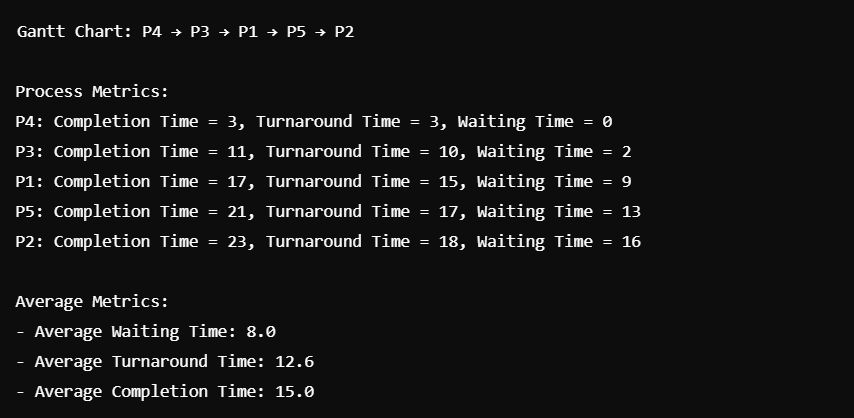
* Processes are executed in the order they arrive. For example, **P4** arrives first and is executed first, followed by **P3**, **P1**, **P5**, and **P2**.

1. **Completion, Waiting, and Turnaround Times:**

* **Completion Time**: The time when each process finishes execution.
* **Turnaround Time**: The time from process arrival to completion. For example, **P4** has a turnaround time of 3 (arrives at 0 and completes at 3).
* **Waiting Time**: The time a process spends waiting in the queue before starting execution. For example, **P5** has a waiting time of 13, as it arrives at 4 but waits until **P1** and **P3** finish.

**Insights on FCFS Results:**

1. **Fair Execution Order:**  
   Processes are executed in the order they arrive. For example, **P4** (arrival = 0) was executed first, and **P2** (arrival = 5) last, maintaining fairness.
2. **Execution Delay:**  
   Longer processes delay shorter ones. For instance, **P3** (burst = 8) caused significant delays, leading to a high waiting time of 16 for **P2**.
3. **High Waiting Times for Later Processes:**  
   Waiting times increase for processes arriving later. For example, **P1** had a waiting time of 9, while **P5** waited 13 units of time.
4. **Turnaround Time Depends on Waiting Time:**  
   The turnaround time is higher for processes with long waiting times. For instance, **P2** had a turnaround time of 18 due to its long wait.
5. **Best Use Case:**  
   FCFS works best when processes have similar arrival and burst times. In this case, processes like **P4** (turnaround = 3) benefit, but others like **P5** (turnaround = 17) are penalized.



**Advantages of FCFS:**

* **Simple**: Easy to implement and understand.
* **Fair**: Processes are executed in arrival order.
* **No Starvation**: All processes are eventually executed.

**Disadvantages of FCFS:**

* **Execution delay**: Long processes delay shorter ones.
* **Non-Preemptive**: A process runs to completion once started.
* **High Waiting Time**: Can lead to poor average waiting time.
* **Inefficient**: Not suitable for time-sensitive tasks.

**Round Robin Algorithm:**

The Round Robin (RR) scheduling algorithm is a preemptive method used primarily in operating systems for process scheduling. It is designed to allocate CPU time to each process in a fair and cyclic manner (quanta), ensuring that every process receives an equal share of CPU resources over time. This algorithm is particularly effective in time-sharing systems, where multiple processes need to be executed concurrently.

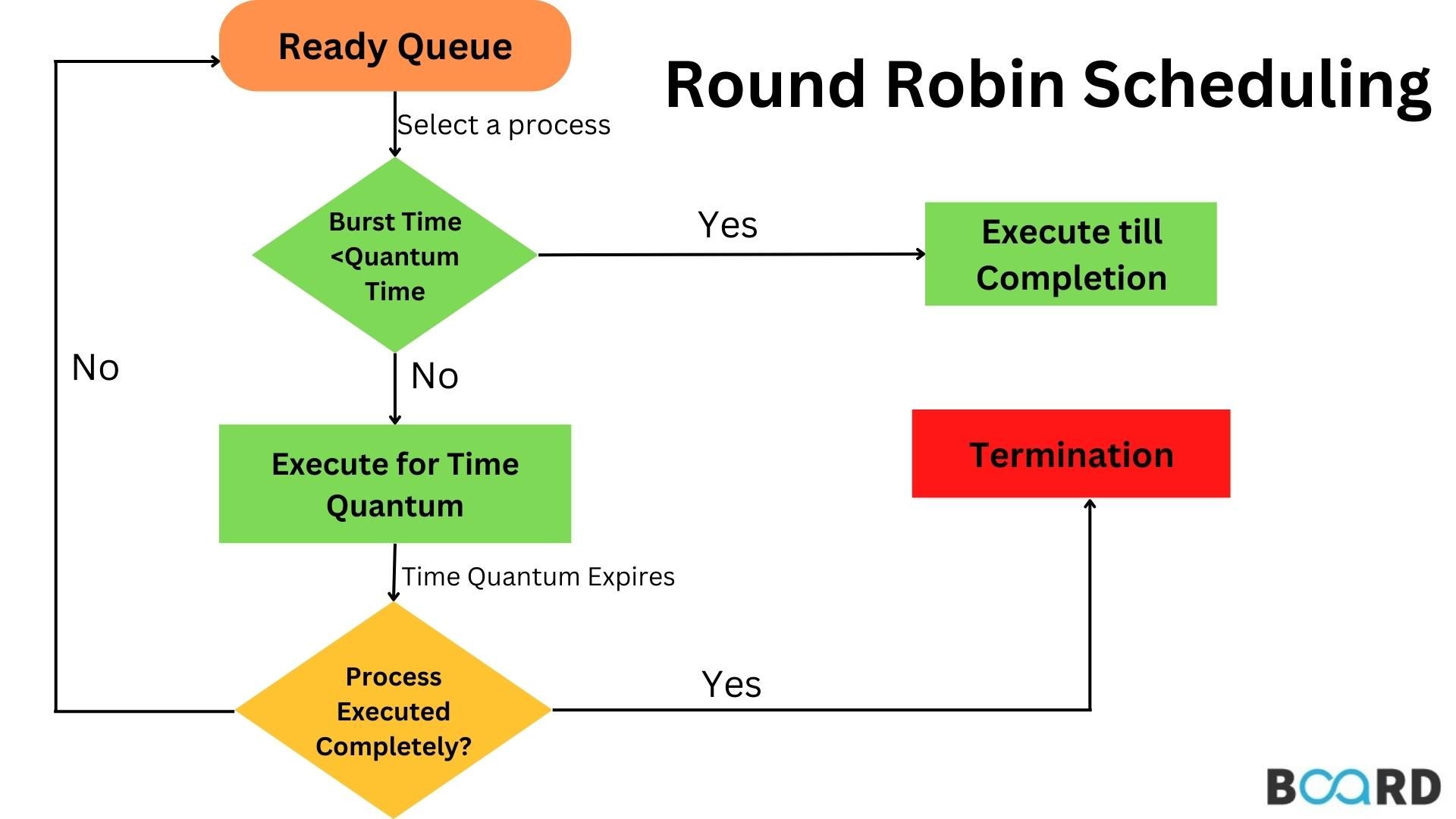
**How does Round Robin work:**

**Process Queue:** All processes that are ready to execute are placed in a queue.

**Time Quantum:** Each process is assigned a fixed time slice known as the time quantum. This is the maximum amount of time a process can run before it is preempted.

**Execution Cycle**: The CPU scheduler picks the first process from the ready queue and allows it to run for the duration of the time quantum. If the process completes its execution within this time, it terminates; otherwise, it is preempted and moved to the end of the queue. The next process in line then gets its turn for execution.

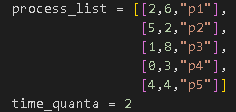
**Cyclic Nature:** This cycle continues until all processes have completed their execution.



**Characteristics of the algorithm:**

* **Preemptive Algorithm:** Allows processes to be interrupted and resumed later.
* **Fairness:** Each process receives an equal opportunity for CPU access.
* **Time Quantum**: The choice of time quantum significantly affects performance; too short leads to high context switching overhead, while too long may lead to poor response times.
* **Context Switching:** Frequent switching between processes can lead to increased overhead.

**The algorithm implementation:**



A list of processes is given with the arrival time and the burst time, assuming a time quantum of 2 seconds.

**The order of execution based on the arrival time:**



**The algorithm results:**

A screenshot of a computer

Description automatically generated

### **Completion Time (Average: 16.4)**

* The average completion time of processes is moderate, indicating a balanced distribution of CPU time.
* Processes with smaller workloads (like p2) complete faster, while larger processes (like p3) take longer.

### **2.** **Turnaround Time (Average: 14):**

* Turnaround time includes both execution and waiting times, and the average indicates that the system handles processes efficiently without significant delays.
* Processes such as p2 have a much lower turnaround time compared to p3 and p1, which reflects their smaller execution requirements.

### **3. Waiting Time (Average: 9.4):**

* Higher waiting times (like p3 and p1) suggest that the time quantum may not be ideal for larger processes.
* Processes with lower waiting times (p2 and p4) benefit from either a smaller workload or their initial placement in the queue.

### **4. Response Time (Average: 1.6):**

* Low response times indicate that the Round Robin algorithm effectively provides quick feedback for all processes, which is one of its key strengths.
* Processes like p4 have a response time of 0, indicating they were executed immediately upon arrival.

**Advantages:**

**Starvation-Free:** Every process eventually gets CPU time, eliminating starvation issues common in priority-based scheduling algorithms.

**Fair Allocation:** Ensures that all processes receive an equitable share of CPU resources.

**Disadvantages:**

**Overhead from Context Switching:** Frequent context switches can degrade system performance.

**Performance Dependency on Time Quantum:** Finding an optimal time quantum can be challenging; it varies based on system workload and process characteristics.

**No Prioritization:** All processes are treated equally, which may not be suitable for all applications, especially those requiring priority handling.

**Priority Scheduling:**

**Priority Scheduling** is an operating system scheduling algorithm that that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

**Pre-emptive Priority Scheduling:**

In Preemptive Priority Scheduling, the CPU always executes the highest-priority process, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

**Non-Pre-emptive Priority Scheduling:**

In Non-Preemptive Priority Scheduling, the CPU executes a process until it finishes, regardless of the arrival of higher-priority processes. New processes must wait in the queue until the current process completes.

**Characteristics of Priority method:**

* A CPU algorithm that schedules processes based on priority.
* If two jobs having the same priority are ready, we use FCFS and Round-Robin to choose between them.
* Used to perform batch processes.
* A number is given to each process to indicate its priority level.

**Pre-emptive Priority Implementation and Results:**

**Current Time:**

The current time keeps track of when the CPU is available for the next process and is updated as each process executes. It starts at 0, and after each execution cycle, the current time increments by 1. If a process finishes its execution, the current time moves to the next available process or idles if no process is ready.

**For example:**

* **P4** starts execution at time 0 and has a burst time of 3, the current time will increment with each cycle until P4 finishes.
* If no process is available, the CPU remains idle, and the current time advances by 1.
* At time 5, process **P2,** which has a higher priority than the other processes currently available, arrived and preempted the process that was previously being executed by the CPU

**Execution Order (Gantt Chart):**

Processes are executed based on their priority at the current time, and the execution order may change dynamically if a higher-priority process arrives.

A black background with colorful dots

Description automatically generated with medium confidence

**Average metrics:**

**Average Completion Time**: 12.0  
The average time taken by all processes to complete is relatively low, considering the 5 processes involved.

**Average Turnaround Time**: 9.6  
On average, it takes 9.6 units of time for a process to complete from the moment it is initiated, reflecting some delays due to preemption, particularly for p3 and p1.

**Average Waiting Time**: 5.0  
On average, processes waited for 5 units of time before being executed, with p3 and p1 contributing significantly to this metric due to their low priority.

**conclusion:**

The preemptive priority scheduling algorithm generally ensures that high-priority processes are executed first, leading to lower completion times for them. However, processes with lower priorities (like p3 and p1) experience longer waiting and turnaround times due to preemption. The average completion, turnaround, and waiting times provide a clear indication of the efficiency and delays caused by preemption in this scheduling method.

**Advantages of Pre-emptive Priority:**

**Efficient Resource Utilization**: High-priority processes get the CPU time they need, ensuring that important tasks are completed quickly, improving overall system efficiency.

**Faster Response Time for Critical Tasks**: Critical or time-sensitive tasks (e.g., real-time systems) are given priority, reducing delays for processes that require immediate attention.

**Fairness**: Lower-priority tasks are preempted only when a higher-priority task arrives, which can be more equitable than non-preemptive scheduling in some cases.

**Adaptability**: It allows the system to adjust dynamically to changing workloads, ensuring that important tasks are not delayed by lower-priority ones.

**Disadvantages of Pre-emptive Priority:**

**Starvation**: Low-priority processes may never get CPU time if high-priority tasks keep arriving, causing some processes to be indefinitely delayed.

**Increased Overhead**: Context switching between tasks (especially for preemption) can incur significant overhead, which may reduce the overall performance of the system.

**Complexity**: Managing priorities and ensuring that tasks are scheduled appropriately can be more complex, requiring sophisticated algorithms to handle priorities efficiently.

A black screen with white text

Description automatically generated

**Non-Pre-emptive Priority Implementation and Results:**

**Current Time:**

The current time tracks when the CPU is available for the next process. Unlike preemptive scheduling, the CPU executes a process to completion before moving to the next one. The current time starts at 0 and is updated by adding the burst time of each completed process.

**For example:**

* **P4** starts execution at time 0 with a burst time of 3. The current time becomes 3 after **P4** completes.
* The next process, **P3**, starts at time 3, and the current time is updated to 11 after **P3** finishes.

**Execution Order (Gantt Chart):**

Processes are selected based on their priority among the available processes at the current time. Lower numerical values indicate higher priority. Once a process is selected, it runs to completion without interruption.

**For example:**

**P4** (priority 2) arrives first and executes completely.

At time 3, **P3** (priority 1) is available and has the highest priority among the ready processes, so it executes next.

A screenshot of a computer

Description automatically generated

**Average metrics:**

**Average Completion Time**: 13.8  
The average completion time reflects the time taken for all processes to finish execution. It is slightly higher due to the sequential execution nature of non-preemptive scheduling.

**Average Turnaround Time**: 11.4  
Turnaround time is influenced by the execution order. High-priority processes keep this average relatively low, while low-priority processes increase it.

**Average Waiting Time**: 6.8  
Processes with lower priority, like p5, p2, and p1, contribute to the relatively higher average waiting time.

**conclusion:**

Non-preemptive priority scheduling is efficient for systems where process priorities rarely change, as it favors high-priority processes. However, it can lead to significant delays for lower-priority processes, affecting overall fairness. The calculated metrics—**Average Completion Time (13.8)**, **Average Turnaround Time (11.4)**, and **Average Waiting Time (6.8)**—highlight the balance between execution efficiency and the limitations of prioritization without preemption.

**Advantages of Non-Pre-emptive Priority:**

**Simple and Easy to Implement:** Non-preemptive scheduling is straightforward, as it doesn’t involve context switching or process interruption. Once a process starts, the CPU handles it to completion, simplifying implementation.

**Predictable Execution:** The lack of interruptions ensures a predictable execution flow, making it suitable for real-time systems where predictability is crucial.

**Efficient for High-Priority Processes:** Processes with higher priorities are executed first, ensuring critical tasks are handled quickly.

**Lower Overhead:** Since no process is interrupted mid-execution, there is no need for frequent context switching, which reduces overhead and improves CPU efficiency.

**Disadvantages of Non-Pre-emptive Priority:**

**Starvation of Low-Priority Processes:** Processes with lower priorities may experience long delays or even indefinite waiting if higher-priority processes keep arriving. This is known as the **"starvation problem."**

**Lack of Flexibility:** The algorithm is not well-suited for dynamic environments where new, urgent processes may arrive. High-priority processes cannot interrupt ongoing lower-priority tasks, potentially delaying critical work.

**Suboptimal Resource Utilization:** If a high-priority process requires I/O or waits for a resource, the CPU may remain idle, leading to inefficient utilization.

**SJF Scheduling:**

Shortest Job First (SJF) scheduling is one of the most widely used CPU scheduling algorithms. It is designed to minimize the average waiting and turnaround times of processes by prioritizing tasks based on their burst times (the time required to complete execution). SJF exists in two variants: Non-Preemptive SJF and Preemptive SJF, also known as Shortest Remaining Time First (SRTF).

**Non-Pre-emptive Shortest Job First (SJF):**

This scheduling algorithm prioritizes processes based on their arrival time and burst time. It first sorts all processes by arrival time and then selects the process with the shortest burst time among those ready to execute. Since it is non-preemptive, once a process starts execution in the CPU, it continues uninterrupted until it completes.

* **Python Implementation**

Coding non-preemptive SJF is very simple. The function **sjf()** is created and accepts a *process\_list*, where each process is represented by a list containing its *burst time, arrival time, and process ID*. The algorithm maintains a clock **(t)** and iteratively processes tasks. At each iteration, it checks for processes available at the current time and selects the one with **the shortest burst time**. If no processes are available, the CPU remains idle until a process arrives. Once a process is selected, it is serviced to completion without interruption. Its completion time, turnaround time, waiting time, and response time are calculated and added to a list, which stores details of all finished processes. The Gantt chart, a visual representation of process execution order, is also updated in this step.

After all processes are executed, the function calculates the **average waiting time**, **average turnaround time**, and **average response time** for the completed processes. It is formatted in a table using the **tabulate** library. This implementation effectively showcases how non-preemptive SJF minimizes waiting time for shorter jobs.A screenshot of a computer

Description automatically generated

**Advantages of Non-Pre-emptive (SJF):**

1. **Minimized Average Waiting Time:**
   * By prioritizing shorter processes, SJF ensures that smaller tasks finish quickly, reducing the time other processes wait in the queue. This leads to a lower average waiting time compared to many other scheduling algorithms.
2. **Fair Treatment to Small Jobs:**
   * SJF favors shorter processes, which might otherwise be delayed significantly in other scheduling algorithms like First-Come-First-Serve (FCFS).
3. **Simple to Implement:**
   * Since SJF operates by statically sorting processes based on their burst times and does not require continuous context switching, its implementation is straightforward.
4. **Efficient for Batch Processing:**
   * In environments where the burst times of jobs are predictable, such as batch processing systems, SJF can significantly improve overall system throughput.

**Disadvantages of Non-Pre-emptive (SJF):**

1. **Convoy Effect**:

* If a long process arrives first, shorter processes will have to wait until the long process is completed, leading to poor overall system responsiveness.

1. **Starvation**:

* Processes with longer execution times may face indefinite delays if shorter processes keep arriving, as SJF always prioritizes the shortest job first.

1. **Difficulty in Predicting Job Length**:

* SJF requires prior knowledge of the execution time of each process, which may not always be feasible or accurate in practical systems.

1. **Not Suitable for Time-Sharing Systems**:

* Non-preemptive SJF does not support process preemption, making it unsuitable forinteractive systems or environments where quick response times are critical.

1. **Inefficient for Mixed Workloads**:

* In systems with a mix of short and long processes, SJF may not efficiently utilize CPU resources, as it continuously prioritizes short processes over longer ones.

**Pre-emptive Shortest Job First (SJF):**

Also known as Shortest Remaining Time First (SRTF), this scheduling algorithm is similar to Non-Preemptive SJF but with the key distinction that it allows interruptions. If a new process arrives with a shorter burst time than the currently running process, the CPU will preempt the ongoing process and allocate resources to the new one. This approach ensures that processes with the shortest remaining execution time are always prioritized, reducing overall waiting time but potentially causing frequent context switching.

* **Preemptive SJF Code Explanation:**
* **preemptive\_sjf(data) Function:**
  + **Input:** data is a list of lists, where each inner list represents a process: [arrival\_time, burst\_time, process\_name].
  + **Sorting:** data.sort(key=lambda x: (x[0], x[1])) sorts the processes based on arrival time first, and then by burst time (in case of ties in arrival times). This initial sorting is important for handling processes arriving at the same time.
  + **Initialization:**
    - current\_time: Tracks the current time in the simulation.
    - completed: Counts the number of completed processes.
    - n: The total number of processes.
    - remaining\_time: A list to keep track of the remaining burst time for each process.
    - gantt\_chart: A list to represent the Gantt chart, storing the process names in the order they execute.
    - waiting\_time, turnaround\_time, response\_time: Lists to store the respective times for each process. response\_time is initialized with -1 to indicate that a process hasn't responded yet.
    - completed\_processes: A list of dictionaries to store detailed information about each completed process.
  + **Main Loop (while completed < n):**
    - **Finding the Shortest Job:** The inner loop iterates through the processes to find the one that has arrived (data[i][0] <= current\_time), has remaining time (remaining\_time[i] > 0), and has the shortest remaining time (remaining\_time[i] < min\_time).
    - **Idle Time:** If no process is available (idx == -1), the current\_time is incremented.
    - **Response Time:** If it's the first time a process is selected, its response time is calculated.
    - **Execution and Preemption:** The selected process's remaining time is decremented. The process name is added to the gantt\_chart.
    - **Process Completion:** When a process finishes (remaining\_time[idx] == 0), its completion time, turnaround time, and waiting time are calculated. The process information is added to completed\_processes.
    - **Time Increment:** The current\_time is incremented after each time unit.
  + **Averages and Output:** The code calculates the average waiting time, turnaround time, and response time. It then prints a formatted table of the process information and the Gantt chart.

A black and white text

Description automatically generated

**Advantages of Pre-emptive (SJF):**

**1. Minimizes Average Waiting Time:**

* By always executing the process with the shortest remaining time, the algorithm ensures that smaller processes complete quickly. This reduces the average waiting time for processes in the ready queue.

**2. Minimizes Turnaround Time:**

* As shorter jobs are prioritized, the overall turnaround time (the time from process arrival to completion) is reduced compared to other scheduling algorithms like FCFS.

**3. Efficient CPU Utilization:**

* By minimizing waiting and turnaround times, the CPU spends less time idle and is utilized more effectively.

**4. Fairness for Short Jobs:**

* Processes with smaller burst times are less likely to be delayed behind long-running processes, which can improve responsiveness for such tasks.

**5. Adaptive for Dynamic Environments:**

* The pre-emptive nature allows the system to adapt dynamically to new processes. For instance, if a shorter process arrives, it immediately pre-empts the currently running process.

**6. Improved System Performance:**

* Prioritizing shorter jobs often leads to quicker overall completion of processes, which can be particularly beneficial in systems where a mix of short and long jobs is common.

**Disadvantages of Pre-emptive (SJF):**

**1. Complex Implementation:**

* Maintaining accurate and real-time knowledge of the burst time for processes is challenging.
* The algorithm requires constant pre-emption and recalculation of remaining burst times, which can be computationally expensive.

**2. Starvation of Longer Processes:**

* Long-running processes may suffer from starvation, especially in systems with a continuous influx of short jobs, as they are repeatedly pre-empted**.**

**3. Overhead of Context Switching:**

* Frequent pre-emption leads to an increased number of context switches, which can result in significant CPU overhead and reduced overall system efficiency.

**4. Prediction Difficulty:**

* The algorithm assumes precise knowledge of burst times, but in real-world scenarios, this information may not be available or accurately predicted.

**5. Not Suitable for Interactive Systems:**

* Interactive systems often require responsiveness to user inputs, but SJF focuses on burst time, which might not align with user-centric priorities.

**6. Unfair to Long Jobs:**

* Long processes are frequently interrupted or delayed, leading to higher waiting and turnaround times for such tasks, which can cause inefficiency in workloads requiring balanced execution.

**7. Increased Scheduling Complexity:**

* Continuous monitoring and sorting of processes based on their remaining burst time add significant complexity to the scheduling mechanism.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Bonus :**

* 1. **A screenshot of a computer

     Description automatically generatedHome page that shows types of CPU scheduling:**

**With a clear view :**

**When you click on a click icon or on the algorithm name from the navbar it will redirect to the page of the algorithm:**

**(all with the same Ui but will be different in the logic of the algorithm)**

**Let’s see the FCFS**

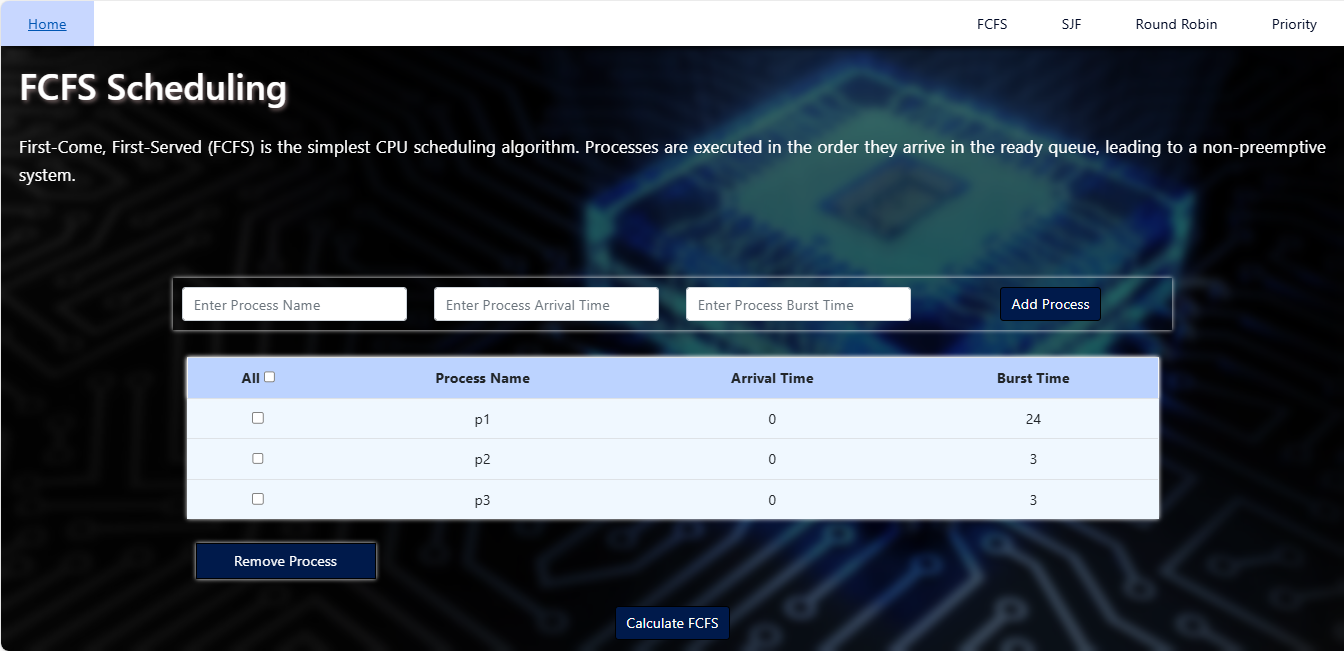
A screenshot of a computer

Description automatically generated

**As shown here is a form that you will enter**

**the “process name” and “arrival time” and “burst time”**

**then click on Add Process button it will add it directly to the table.**

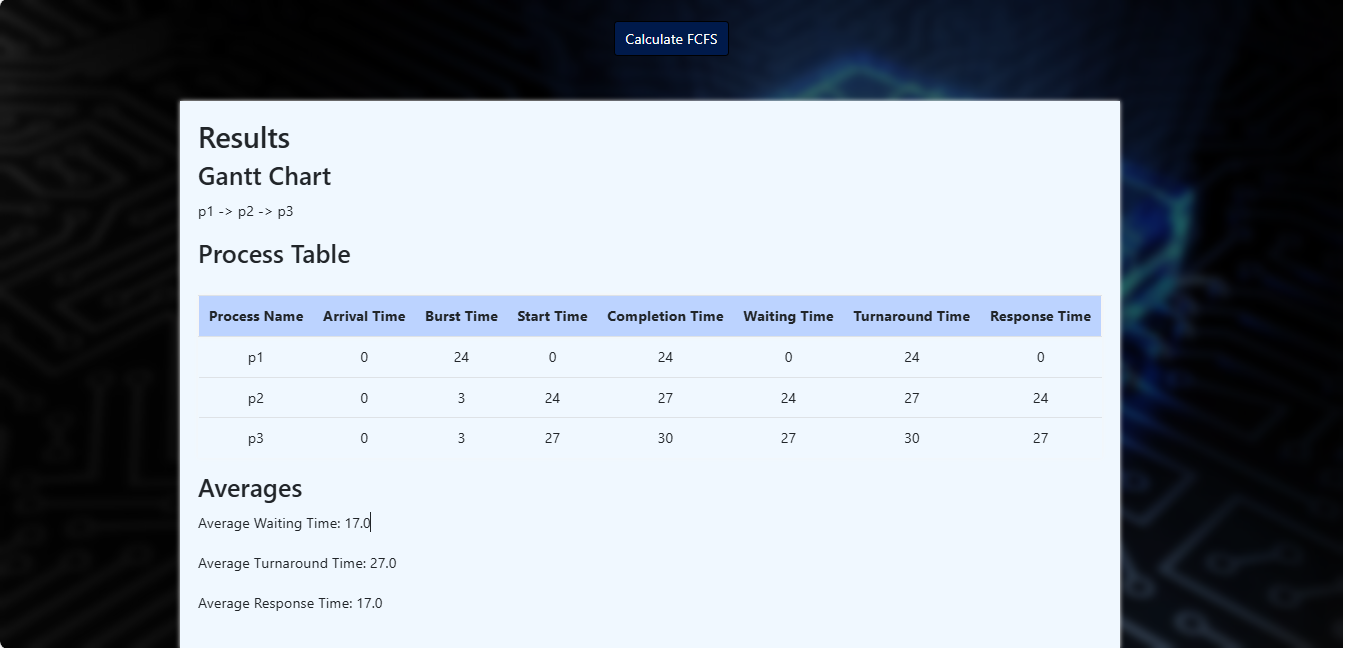


**Now when you click on Calculate FCFS it will fetch the data from the table and send it as a list to be python code that apply the FCFC algorithm on these lists then return the result as shown here will create a new table contains our main data and the calculations of :**

* + - **Start time**
    - **Completion time**
    - **Waiting time**
    - **Turnaround time**
    - **Response time**

**Then will compute the average of each**

* + - **Average Waiting time**
    - **Average Turnaround time**
    - **Average Response time**



**And the rest of the algorithms with the same steps with small differences like**

**Priority scheduling we add a input field to enter the priority of the process with the rest of data to calculate depends on it.**

Thank you..