CPU SCHEDULING

A MINI PROJECT REPORT

18CSE356T - Distributed Operating Systems

Submitted by
JAGATHGURU[RA2111003011590]
KIRAN [RA2111003011582]
VISHNU VARADHAN [RA2111003011600]

Under the guidance of

Dr.Vinod D
Assistant Professor, Department of Computing Technologies

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

S.R.M. Nagar, Kattankulathur, Chengalpattu District

AUG 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that Mini project report titled **"CPU SCHEDULING "** is the bonafide work of , **JAGATHGURU[RA2111003011590], KIRAN[RA2111003011582],VISHNU[RA2111003011600]** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE	**SIGNATURE**
DR.VINOD.D	Dr. M. Pushpalatha
Assistant Professor	**HEAD OF THE DEPARTMENT**
Department of Computing Technologies	Professor & Head
	Department of Computing Technologies

ABSTRACT

The efficient allocation of CPU resources is a cornerstone of modern computing systems. CPU scheduling, as a core component of operating systems, orchestrates the execution of processes or threads, ultimately impacting system performance and user experience. This project delves deeply into the world of CPU scheduling, offering a comprehensive exploration of various scheduling algorithms and their implications. We conduct in-depth simulations and analyze real-world case studies to assess how scheduling algorithms perform under diverse scenarios, from basic round-robin to complex priority-based schemes.

Our study investigates the intricate balance between system goals, such as optimizing CPU utilization, minimizing turnaround times, and ensuring fairness among competing processes. Through detailed experiments and performance evaluations, we provide empirical insights into the strengths and limitations of different scheduling strategies.

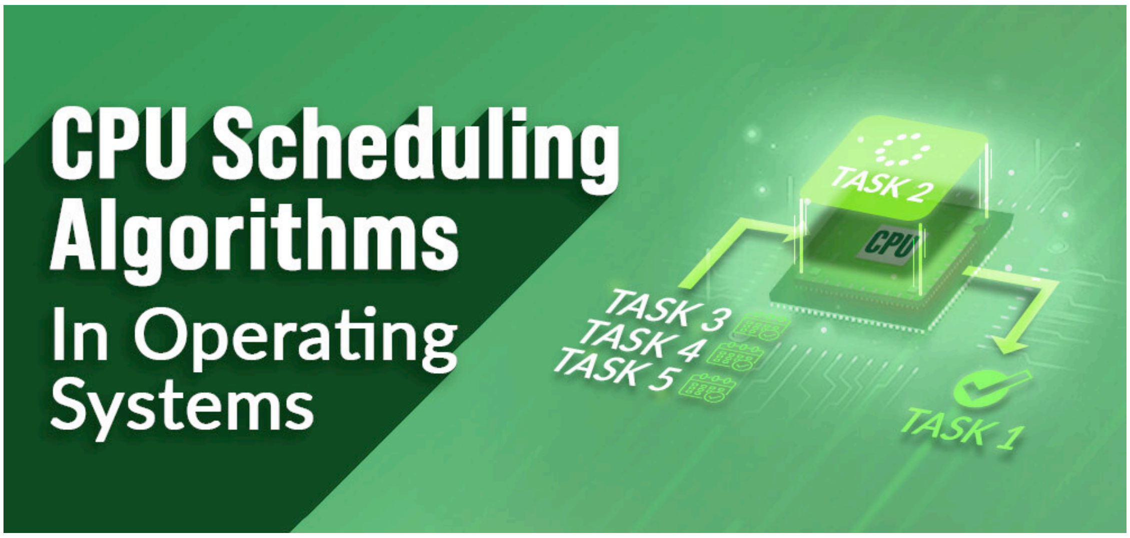
The project illuminates the trade-offs involved in CPU scheduling, elucidating the challenges of maintaining responsiveness while maximizing resource utilization.

This research is not only an academic exercise but a practical exploration of the systems that power our digital world. By understanding CPU scheduling intricacies, we aim to inform future decisions in system design, cloud computing, and high-performance computing environments.

In sum, this project underscores the enduring significance of CPU scheduling in enhancing overall system efficiency, a vital concern in the ever-evolving landscape of computing.

TABLE OF CONTENTS

1.**Problem Statement**



The efficient allocation of central processing unit (CPU) resources is a critical concern in operating system design and performance optimization. In contemporary computing environments, where multiple processes or threads contend for CPU time, the choice of a CPU scheduling algorithm profoundly influences system performance and user experience. The primary challenge lies in selecting or designing scheduling algorithms that strike a balance between various objectives, such as maximizing CPU utilization, minimizing response times, ensuring fairness, and adapting to dynamic workloads.

This project addresses the complex problem of CPU scheduling and the necessity of choosing the most appropriate scheduling algorithm for specific computing scenarios. We aim to analyze and compare the performance of four fundamental CPU scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job Next (SJN), Priority Scheduling, and Round Robin. By doing so, we seek to uncover how each algorithm impacts system performance, resource utilization, and fairness in task execution.

Furthermore, this project confronts the challenge of aligning the theoretical underpinnings of CPU scheduling algorithms with practical application in real-world operating systems. Understanding the intricacies of each algorithm and their real-world relevance is vital for system designers and administrators as they strive to optimize system performance and user satisfaction.

In summary, the problem statement encapsulates the core challenge of CPU scheduling in the context of diverse objectives and a multitude of scheduling algorithms, which our project seeks to address through simulation, analysis, and comparative evaluation.

2. Introduction

In the ever-evolving landscape of computing, one constant remains: the need for efficient and equitable allocation of central processing unit (CPU) resources. CPU scheduling, a cornerstone of modern operating systems, serves as the linchpin that orchestrates the execution of processes or threads, ultimately influencing system performance and user experience.

The significance of CPU scheduling arises from the fact that computer systems are multitasking environments, where numerous processes concurrently vie for access to the CPU. This environment demands a methodical approach to decide the order in which processes are granted CPU time, ensuring fairness, efficiency, and responsive execution.

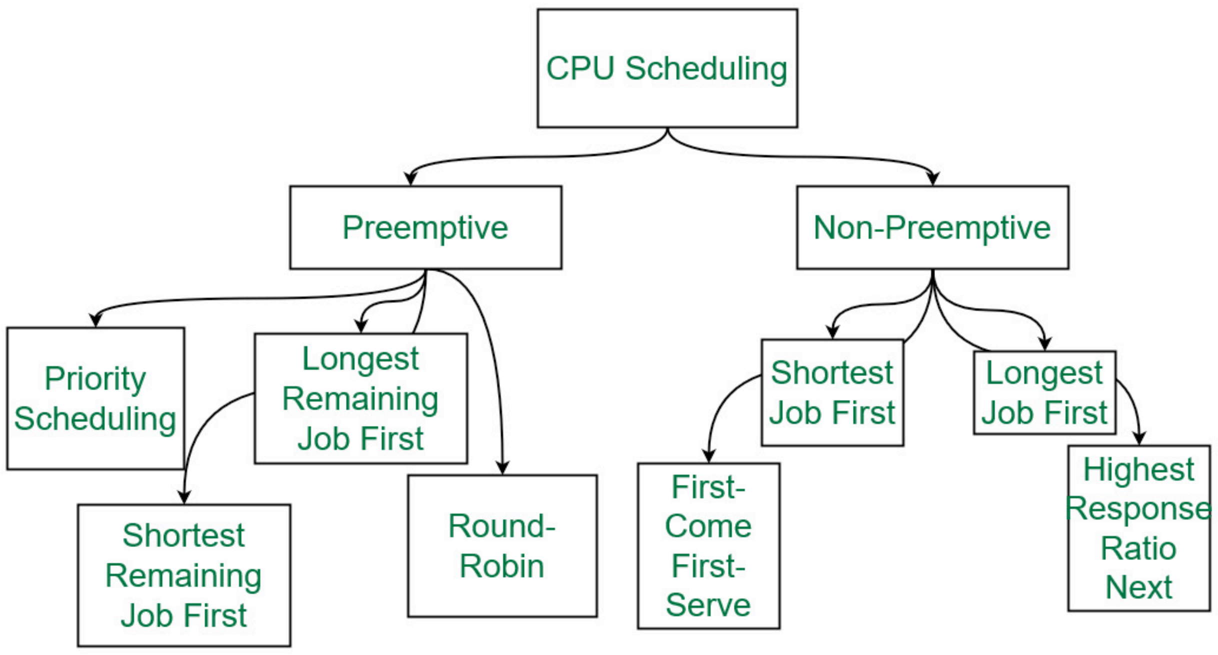
The objectives of this project are twofold: first, to delve deep into the world of CPU scheduling algorithms, and second, to shed light on their real-world applications and implications. We embark on a comprehensive exploration of four fundamental CPU scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job Next (SJN), Priority Scheduling, and Round Robin. Each of these algorithms embodies unique characteristics, designed to address specific system objectives.
The core challenge underlying CPU scheduling is the effective management of system resources, with the following goals in mind:

- Maximizing CPU utilization to ensure efficient resource usage.
- Minimizing response times to enhance user experience.
- Ensuring fairness in the allocation of CPU time among competing processes.
- Adapting to dynamic workloads and varying system priorities.

Through meticulous simulations and comparative analyses, this project endeavors to scrutinize the strengths and limitations of these scheduling algorithms. By evaluating their performance under a range of scenarios, we aim to offer insights into the contexts where each algorithm excels and where it falls short.
The project doesn't stop at the theoretical aspects of CPU scheduling. It also ventures into the realm of practical applications by examining how these algorithms are implemented in real-world operating systems like Windows, Linux, and macOS. Bridging the gap between theory and practice, we seek to understand the principles that drive contemporary computing environments.

In sum, the journey we undertake is not merely an academic exercise but a practical exploration of the systems that power our digital world. By navigating this rich tapestry of CPU scheduling, we hope to empower system designers, administrators, and anyone intrigued by the inner workings of computing systems with the knowledge to make informed decisions that optimize system performance and resource utilization.

3. Background



The field of CPU scheduling revolves around the intricate orchestration of processes and threads competing for access to the central processing unit (CPU) in modern computer systems. To appreciate the nuances of CPU scheduling and its critical role in computing, it is essential to grasp several fundamental concepts and the context in which scheduling algorithms operate.

Processes and Threads:

- A process is a fundamental unit of execution in an operating system. It encompasses an application's code, data, and system resources, making it an independent entity with its own memory space.
- Within processes, multiple threads can run concurrently. Threads are lighter-weight units of execution within a process, sharing the same memory space and resources, enabling more fine-grained parallelism.

Multiprogramming and Multitasking:

- Multiprogramming refers to the practice of loading multiple programs into memory simultaneously. This enables the CPU to switch between executing different processes or threads rapidly.
- Multitasking extends the concept of multiprogramming, allowing a single CPU to serve multiple users or applications by interleaving their execution.

Scheduling Queues:

- In the CPU scheduling context, processes or threads are often placed in scheduling queues. The Ready Queue contains all processes or threads that are ready to execute.
- The scheduling algorithm selects processes or threads from the Ready Queue for execution based on predefined criteria.

Scheduling Objectives:

- CPU scheduling algorithms aim to balance various system objectives, including:
- Maximizing CPU utilization to keep the CPU busy and productive.
- Minimizing turnaround time, the time taken for a process to complete.
- Reducing waiting times, which can lead to improved user experience.
- Ensuring fairness in CPU allocation among competing processes.
- Adapting to dynamic workloads and system priorities.

Preemptive vs. Non-Preemptive Scheduling:

- Preemptive scheduling allows the CPU scheduler to interrupt a running process or thread and allocate the CPU to another process based on the scheduling criteria.
- Non-preemptive scheduling, in contrast, only allows the CPU to be allocated when the currently executing process voluntarily relinquishes it.

Scheduling Algorithms:

- Various CPU scheduling algorithms have been developed to address these objectives. Common algorithms include:
- First-Come, First-Served (FCFS): Prioritizes processes based on their arrival time.
- Shortest Job Next (SJN): Schedules the process with the shortest expected execution time.
- Priority Scheduling: Assigns priorities to processes, and the CPU scheduler selects the highest-priority process.
- Round Robin: Allocates the CPU in a circular manner, allowing each process a time quantum to execute.

Real-World Significance:

- CPU scheduling is not a mere theoretical exercise; it plays a pivotal role in the real-world performance of operating systems, servers, and applications.
- Effective CPU scheduling is vital in systems ranging from personal computers to data centers and cloud computing platforms.

Understanding these foundational concepts sets the stage for a deeper exploration of CPU scheduling algorithms, their strengths, limitations, and their practical application in the realm of operating systems and system optimization.

4. First-Come, First-Served (FCFS) Scheduling

In the realm of CPU scheduling algorithms, First-Come, First-Served (FCFS) stands as one of the most straightforward and intuitive approaches. As its name suggests, FCFS allocates the central processing unit (CPU) to processes in the order they arrive in the ready queue. The first process to arrive is the first to execute, and the subsequent processes follow the same sequence. FCFS is a non-preemptive algorithm, which means that once a process begins executing, it retains control of the CPU until it completes or voluntarily yields it.

Algorithm Overview:
FCFS operates on a simple principle: the process that arrives first is served first. Here's a step-by-step overview of the FCFS algorithm:

When a process enters the ready queue, it is placed at the end of the queue.
The CPU scheduler selects the process at the front of the queue for execution.
The selected process continues executing until it finishes its task, at which point the next process in the queue is granted access to the CPU.

Key Characteristics:

Simplicity: FCFS is one of the easiest scheduling algorithms to understand and implement. It follows an uncomplicated logic that resembles how tasks are handled in many real-life scenarios, such as waiting in line at a grocery store.

Fairness: FCFS inherently provides fairness, as processes are served in the order they arrive. It ensures that each process gets a chance to execute without any preference or bias.

Suitability for Uniform Workloads: FCFS performs well when processes have roughly equal execution times. In such scenarios, it minimizes waiting times, as no process remains in the ready queue for an extended period.

Limitations:
While FCFS has its merits, it is not without limitations:

Inefficient for Varying Execution Times: FCFS does not take into account the execution time of processes. Consequently, it can lead to inefficient CPU utilization when short processes are held up behind long-running ones. This situation results in increased turnaround times, affecting system responsiveness.

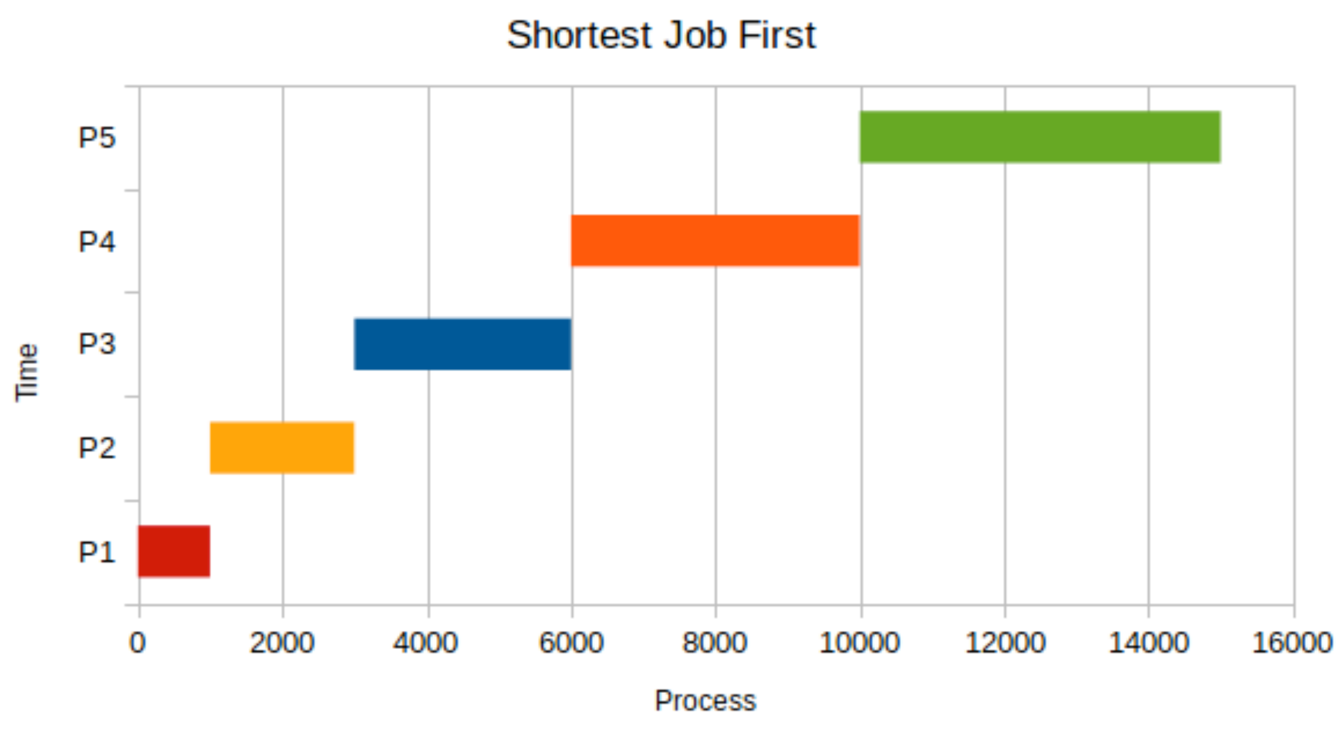
Inflexibility: FCFS is highly inflexible and does not adapt to changing workloads or prioritize more critical processes. It adheres strictly to the arrival order.

Example:
Let's consider a simple example with three processes: A, B, and C, arriving in that order.

Process A arrives first and gains access to the CPU.
Process B arrives while A is still executing, so it joins the queue.
Once A completes its execution, B takes its turn.
Process C arrives while B is running, so it waits.
When B finishes, C is scheduled to run.
Simulation and Results:
Through simulations and analysis, the performance of FCFS can be evaluated in different scenarios. Key metrics, such as CPU utilization, turnaround time, waiting time, and response time, can be measured to assess the algorithm's effectiveness.

Conclusion:
FCFS, while straightforward and fair, may not be the optimal choice in all situations due to its inefficiency when process execution times vary widely. Making an informed decision about selecting a scheduling algorithm requires careful consideration of the specific characteristics of the computing environment and the objectives you aim to achieve.

5.Shortest Job First (SJF) Scheduling



Shortest Job First (SJF) Scheduling is a CPU scheduling algorithm designed to minimize the total execution time of processes. Also known as Shortest Job Next (SJN), SJF prioritizes the process with the shortest expected execution time, often referred to as the "burst time." By scheduling the shortest jobs first, SJF aims to reduce overall execution time and optimize system performance.
Algorithm Overview:

The SJF algorithm operates on the principle of minimizing the expected burst time of processes. Here's an overview of how SJF works:

- When a process enters the ready queue, the scheduler compares the burst time (execution time) of the incoming process with the burst times of processes already in the queue.
- The process with the shortest remaining burst time is selected to run next. This means that processes with shorter tasks are given priority.
- The selected process continues executing until it completes or voluntarily yields the CPU.
- The cycle repeats as new processes arrive, and the algorithm continuously selects the shortest remaining burst time process.

Preemptive vs. Non-Preemptive SJF:
SJF can be implemented in two variations:

- In **preemptive SJF**, if a new process arrives with a shorter burst time than the currently executing process, the CPU scheduler may preempt the running process and allocate the CPU to the newly arrived, shorter process.
- In **non-preemptive SJF**, once a process starts executing, it continues until completion, regardless of any shorter burst time processes that may arrive.

Key Characteristics:

- **Optimality:** In the non-preemptive version, SJF is known to be an optimal scheduling algorithm for minimizing average turnaround time. It schedules processes in a way that minimizes the total execution time, assuming it has knowledge of the exact burst times.
- **Efficiency:** SJF is efficient when processes have widely varying execution times, as it prioritizes short processes, leading to faster turnaround times.

Limitations:

SJF, despite its merits, has a few limitations:

- **Unrealistic Assumption:** The algorithm assumes that the burst times of processes are known in advance. In practice, accurately predicting burst times is challenging.
- **Starvation:** Longer processes may suffer from starvation if many short processes continually arrive, as the short processes are continually prioritized.

Example:

Consider a scenario with three processes: A, B, and C, with their respective burst times (execution times).

- Process A has a burst time of 4 units.
- Process B has a burst time of 2 units.
- Process C has a burst time of 8 units.

Assuming a non-preemptive SJF algorithm:

- Process B would be the first to run, as it has the shortest burst time.
- After B completes, Process A is scheduled.
- Finally, Process C is scheduled, as it has the longest burst time.

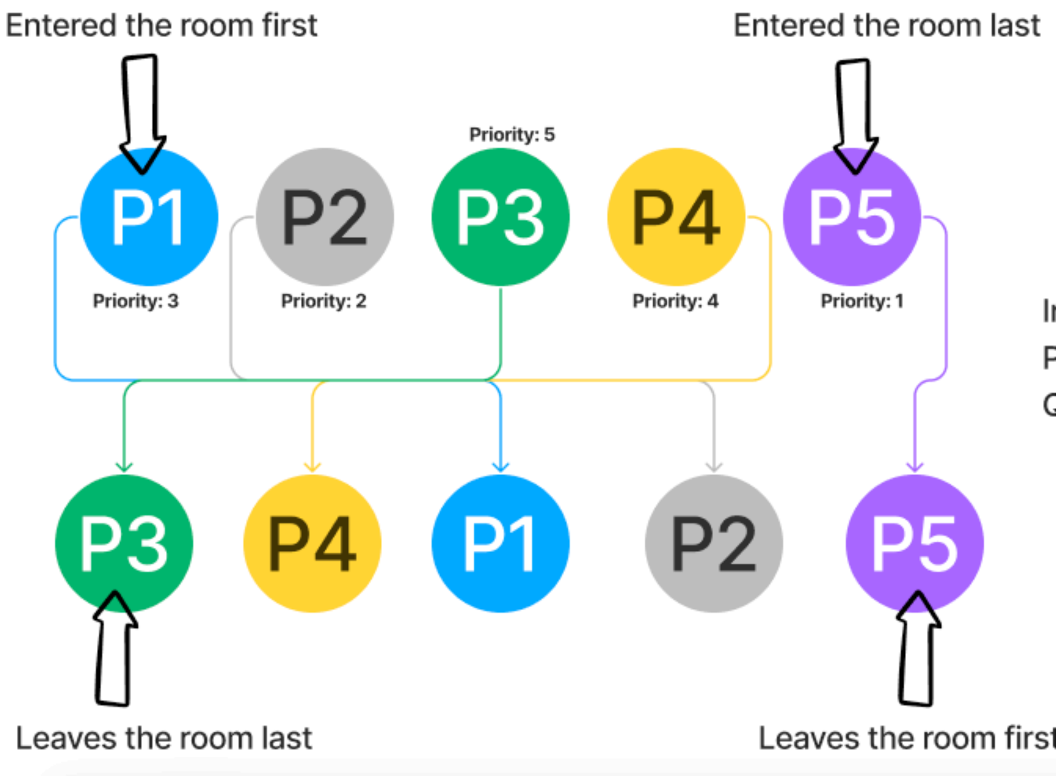
Simulation and Results:

Through simulations and analysis, the performance of SJF can be evaluated in different scenarios. Key metrics, such as CPU utilization, turnaround time, waiting time, and response time, can be measured to assess the algorithm's effectiveness.

Conclusion:

SJF is a promising scheduling algorithm for minimizing average turnaround time when burst times are known. However, in real-world scenarios, accurately predicting burst times is challenging, and longer processes may experience starvation. Selecting the most suitable scheduling algorithm depends on the specific characteristics of the workload and the objectives you aim to achieve in the computing environment.

6.Priority Scheduling



Priority Scheduling is a CPU scheduling algorithm that assigns priorities to each process and allocates the CPU to the process with the highest priority. This approach allows for the execution of critical processes first, ensuring that high-priority tasks are addressed promptly. Priority can be determined by various factors, such as process characteristics, deadlines, or user-defined priorities.

Algorithm Overview:
The Priority Scheduling algorithm is based on the concept of prioritization. Here's an overview of how Priority Scheduling works:

- Each process is assigned a priority value. Higher values often indicate higher priority.
- The CPU scheduler selects the process with the highest priority to run. In the case of multiple processes with the same priority, the algorithm may employ other scheduling criteria, such as First-Come, First-Served (FCFS) or Round Robin.
- The selected process continues to execute until it completes its task or yields the CPU.

Preemptive vs. Non-Preemptive Priority Scheduling:
Priority Scheduling can be implemented in either a preemptive or non-preemptive manner:

- In **preemptive Priority Scheduling**, if a new process with higher priority arrives, the currently executing process may be preempted, and the CPU is allocated to the higher-priority process.
- In **non-preemptive Priority Scheduling**, once a process starts executing, it continues until completion, regardless of the arrival of higher-priority processes.

Key Characteristics:

- **Customization:** Priority values can be assigned based on a variety of factors, allowing for customization to suit specific system requirements. For example, real-time systems may assign priorities based on deadlines.
- **Responsive to Critical Tasks:** Priority Scheduling is particularly useful for addressing time-sensitive or mission-critical tasks promptly. Processes with higher priorities are executed without unnecessary delays.
- **Optimizing for Specific Objectives:** By adjusting the priority values, the algorithm can be tailored to optimize specific objectives, such as minimizing response time or ensuring fairness.

Limitations:

Priority Scheduling has some limitations to consider:

- **Starvation:** Lower-priority processes may experience starvation if higher-priority processes frequently arrive.
- **Inefficient Resource Utilization:** If processes with high priority are continuously arriving, lower-priority processes may receive inadequate CPU time.

Example:

Consider a scenario with three processes: A, B, and C, each assigned a priority level.

- Process A has the highest priority (Priority 1).
- Process B has medium priority (Priority 2).
- Process C has the lowest priority (Priority 3).

Assuming a non-preemptive Priority Scheduling algorithm:

- Process A, the highest priority, will execute first.
- After A completes, Process B will be allocated the CPU.
- Once B finishes, Process C, the lowest priority, will be scheduled.

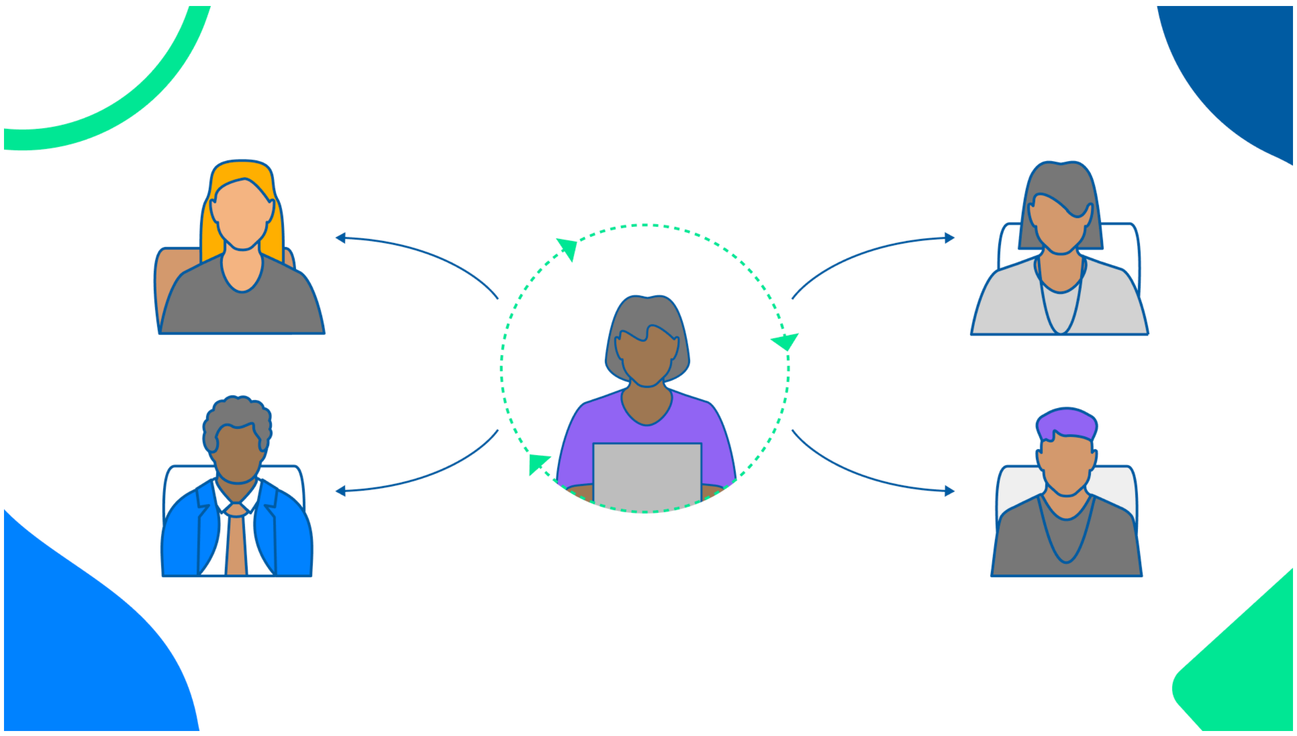
Simulation and Results:

Through simulations and analysis, the performance of Priority Scheduling can be evaluated in different scenarios. Key metrics, such as CPU utilization, turnaround time, waiting time, and response time, can be measured to assess the algorithm's effectiveness.

Conclusion:

Priority Scheduling is a versatile algorithm that allows for the prioritization of processes to meet specific system requirements. By assigning priorities, it is possible to address critical tasks promptly. However, managing priorities effectively and avoiding starvation is a critical consideration when implementing this algorithm. The choice of scheduling algorithm depends on the specific goals and workload characteristics of the computing environment.

7.Round Robin (RR) Scheduling



Round Robin (RR) Scheduling is a widely used CPU scheduling algorithm designed to provide fair and time-sliced execution of processes. In RR, each process is allocated a fixed time quantum or time slice, during which it runs. Once a process's time quantum expires, it is moved to the back of the ready queue, allowing the next process in line to execute. RR is a preemptive scheduling algorithm, ensuring that all processes receive a fair share of CPU time.

Algorithm Overview:
The Round Robin algorithm operates based on the concept of time slicing. Here's an overview of how RR works:

- Each process is assigned a time quantum, denoting the maximum amount of time it can run without being preempted.
- Processes enter the ready queue in the order they arrive.
- The CPU scheduler selects the first process in the queue to run and allocates it the CPU for the time quantum.
- When a process's time quantum expires, it is moved to the back of the queue.
- The cycle repeats, and the next process in line is allocated the CPU for its time quantum.

Key Characteristics:

- **Fairness:** Round Robin provides fairness as each process receives equal CPU time in a round-robin fashion. This ensures that no process is starved of CPU time for an extended period.
- **Responsiveness:** RR is suitable for interactive systems, as it guarantees that each process can be scheduled promptly. Even long-running processes are periodically preempted.
- **Predictable Behavior:** The time quantum can be adjusted to control the trade-off between fairness and system responsiveness.

Limitations:

Round Robin Scheduling has some limitations:

- **Inefficient for Short Processes:** RR may be inefficient for processes with very short burst times, as the overhead of context switching can become a significant portion of the execution time.
- **Long Waiting Times:** Longer processes may experience long waiting times if they need to wait for their turn in the queue, especially if the time quantum is relatively short.

Example:

Consider a scenario with three processes: A, B, and C, and a time quantum of 4 units.

- Process A starts execution but is preempted after 4 units.
- Process B is scheduled to run and also runs for 4 units.
- Process C enters the queue but is moved to the back after 4 units.
- The cycle repeats, and each process is allocated the CPU in a round-robin fashion.

Simulation and Results:

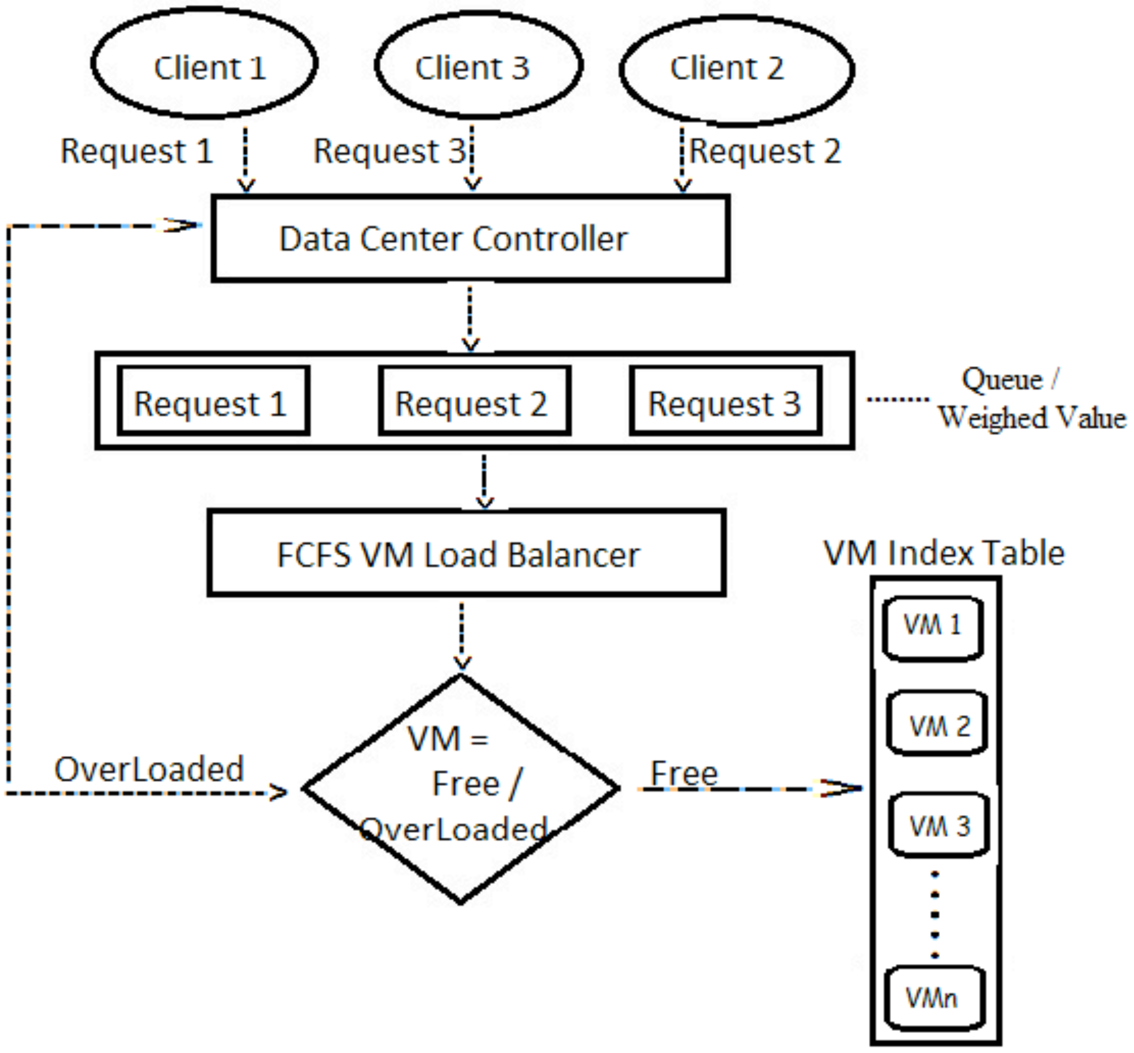
Through simulations and analysis, the performance of Round Robin Scheduling can be evaluated in different scenarios. Key metrics, such as CPU utilization, turnaround time, waiting time, and response time, can be measured to assess the algorithm's effectiveness.

Conclusion:

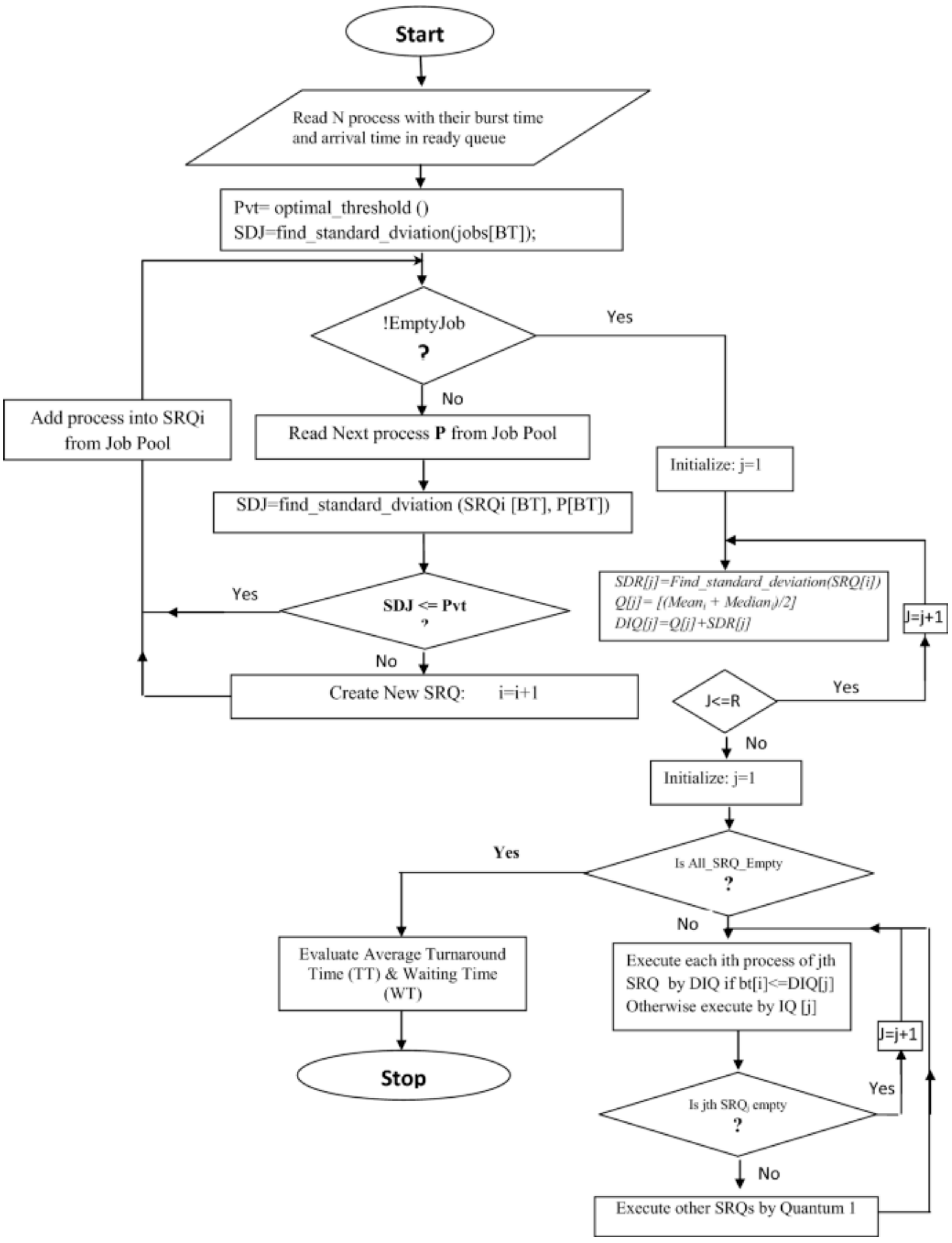
Round Robin Scheduling offers a balanced approach to CPU allocation by providing fairness and responsiveness. It is particularly suitable for interactive systems and environments where processes have varying execution times. However, the choice of the time quantum is critical in determining the algorithm's efficiency and system responsiveness. The specific objectives and workload characteristics of the computing environment play a significant role in selecting the appropriate scheduling algorithm.

8.FLOW CHART DIAGRAMS

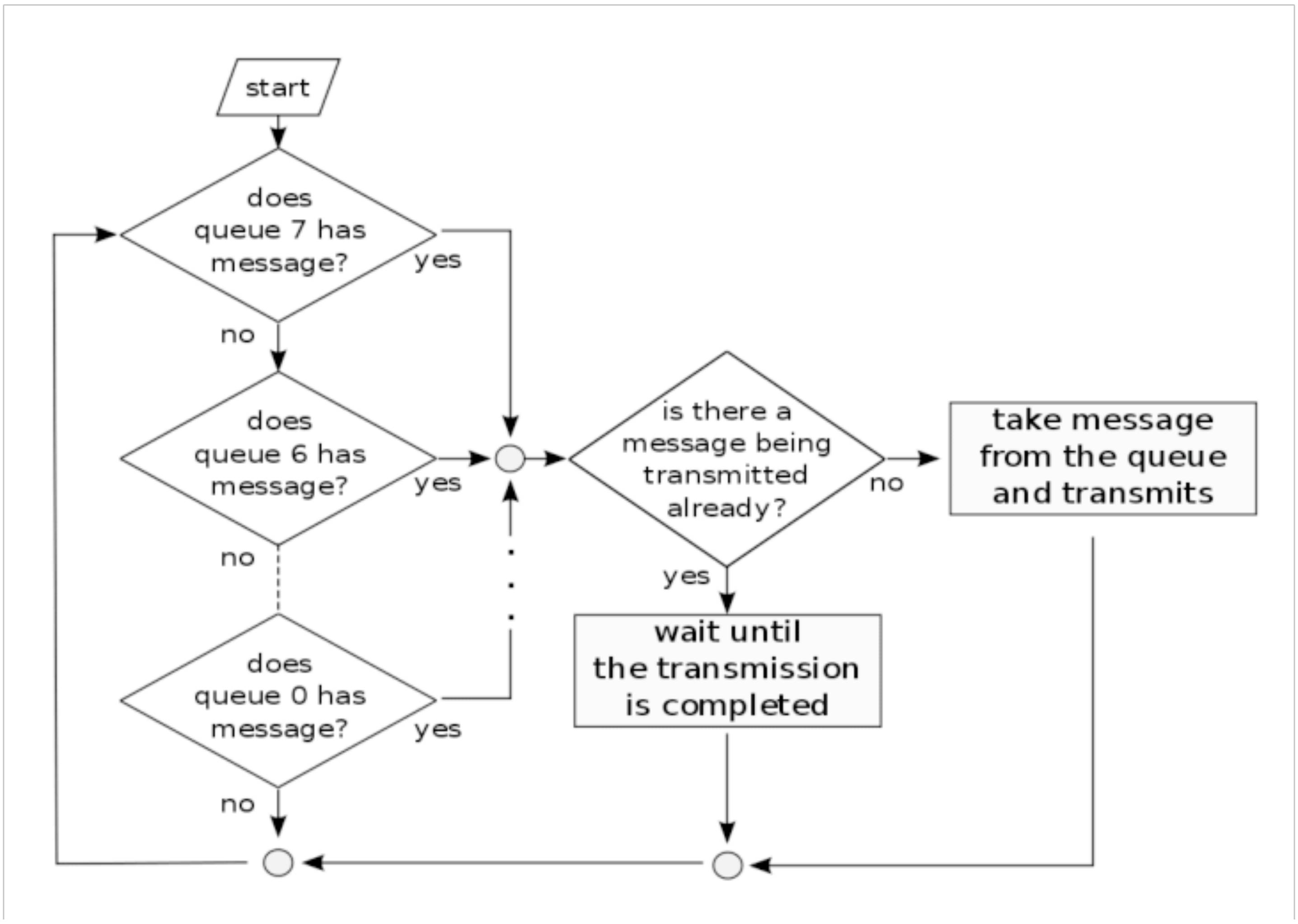
FCFS:



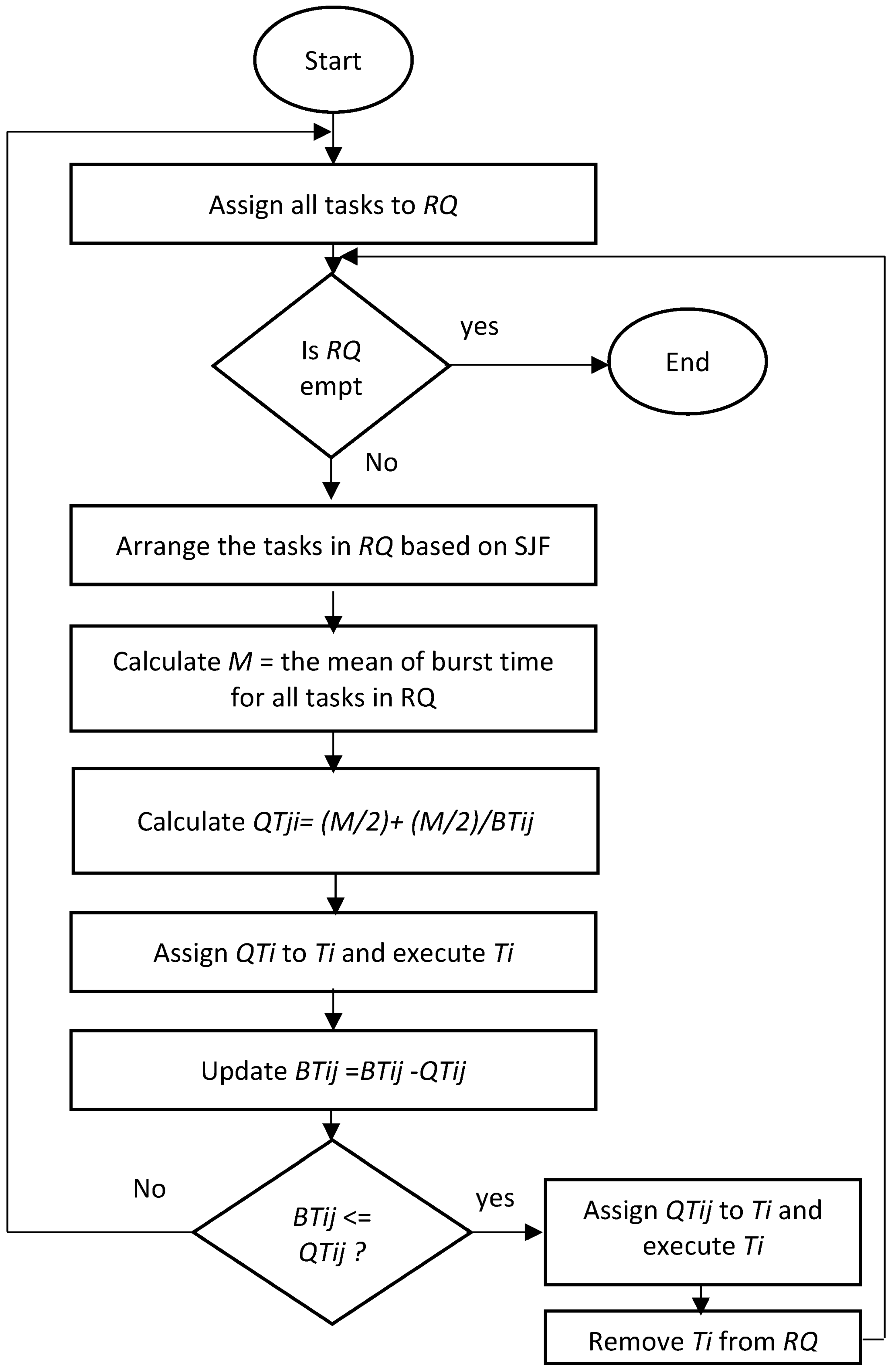
Shortest Job First (SJF) Scheduling:


Start
Read N process with their burst time and arrival time in ready queue
Pvt= optimal_threshold ()
SDJ=find_standard_dviation(jobs[BT]);
!EmptyJob ?
Yes
No
Read Next process P from Job Pool
SDJ=find_standard_dviation (SRQi [BT], P[BT])
SDJ <= Pvt ?
Yes
No
Add process into SRQi from Job Pool
Create New SRQ: i=i+1
Initialize: j=1
SDR[j]=Find_standard_deviation(SRQ[i])
Q[j]= [(Mean_i + Median_i)/2]
DIQ[j]=Q[j]+SDR[j]
J=j+1
J<=R
Yes
No
Initialize: j=1
Is All_SRQ_Empty ?
Yes
No
Evaluate Average Turnaround Time (TT) & Waiting Time (WT)
Stop
Execute each ith process of jth SRQ by DIQ if bt[i]<=DIQ[j] Otherwise execute by IQ [j]
Is jth SRQ_j empty ?
Yes
No
J=j+1
Execute other SRQs by Quantum 1

Priority Scheduling:


start
does queue 7 has message?
yes
no
does queue 6 has message?
yes
no
does queue 0 has message?
yes
no
is there a message being transmitted already?
no
yes
take message from the queue and transmits
wait until the transmission is completed

Round Robin (RR) Scheduling:



Start
Assign all tasks to *RQ*
Is *RQ* empt
yes
End
No
Arrange the tasks in *RQ* based on SJF
Calculate *M* = the mean of burst time for all tasks in RQ
Calculate *QTji= (M/2)+ (M/2)/BTij*
Assign *QTi* to *Ti* and execute *Ti*
Update *BTij =BTij -QTij*
No
BTij <= QTij ?
yes
Assign *QTij* to *Ti* and execute *Ti*
Remove *Ti* from *RQ*

9.Hardware/Software requirements

Hardware Requirements:

- Computer or Server:
  - Any modern computer or server should be sufficient for running a CPU scheduling project. The specific hardware requirements will depend on the complexity of your project and the algorithms you plan to implement.
- Processor (CPU):
  - A multi-core CPU is beneficial, especially if you intend to simulate multiple processes concurrently. It allows for better parallelism and performance.
- Memory (RAM):
  - A minimum of 4GB of RAM is recommended for basic simulations. More RAM may be required for handling a large number of processes or complex scheduling scenarios.
- Storage:
  - Sufficient storage space for the project files, simulation results, and any required datasets. An SSD (Solid State Drive) can help improve overall system responsiveness.
- Input/Output Devices:
  - Standard input/output devices, including a keyboard, mouse, and display, are necessary for user interaction and data input.
- Operating System:
  - The project can be developed and run on various operating systems, including Windows, Linux, or macOS. Ensure compatibility with your chosen development environment and tools.

Software Requirements:

- Programming Language:
  - You will need a programming language for implementing the CPU scheduling algorithms and building the project. Common choices include:
  - Python: Python is a versatile language with a wide range of libraries for simulations and data visualization.
  - C/C++: These languages provide more low-level control and are suitable for performance-critical simulations.
  - Java: Java is a platform-independent language often used for simulations and educational projects.
- Integrated Development Environment (IDE):
  - Choose an IDE that supports your chosen programming language. Popular options include:
  - Google Collab
  - PyCharm (for Python)
  - Eclipse (for Java)
  - Code::Blocks (for C/C++)
- Version Control:
  - A version control system such as Git can help manage project versions and collaborate with others if needed. Platforms like GitHub and GitLab are commonly used for hosting code repositories.

- Libraries/Frameworks:
  - Depending on your programming language, you may need libraries or frameworks for GUI development (if you're building a graphical user interface) and data visualization. For Python, libraries like Matplotlib and Tkinter are commonly used.
- Database (optional):
  - If your project involves data storage and retrieval, you may need a database system. Popular choices include MySQL, PostgreSQL, or SQLite.
- Simulation Tools (optional):
  - Depending on the level of detail in your simulation, you might need specialized simulation tools or platforms. These tools can help with modeling and visualization of scheduling scenarios.
- Documentation and Collaboration Tools:
  - Consider using documentation tools like Markdown for project documentation and communication tools like Slack or Microsoft Teams for team collaboration.
- Virtualization (optional):
  - If you plan to run simulations on different operating systems or environments, virtualization software like VirtualBox or VMware can be helpful.
- Text Editor (for script editing):
  - A text editor, like Notepad++ or Sublime Text, can be handy for editing script files or configuration files.
- Web Server (if needed):
  - If you plan to deploy any web-based components for your project, you might need a web server environment, such as Apache or Nginx.

10.Implementation- Code snippet/ Results- Screen Shots of Output

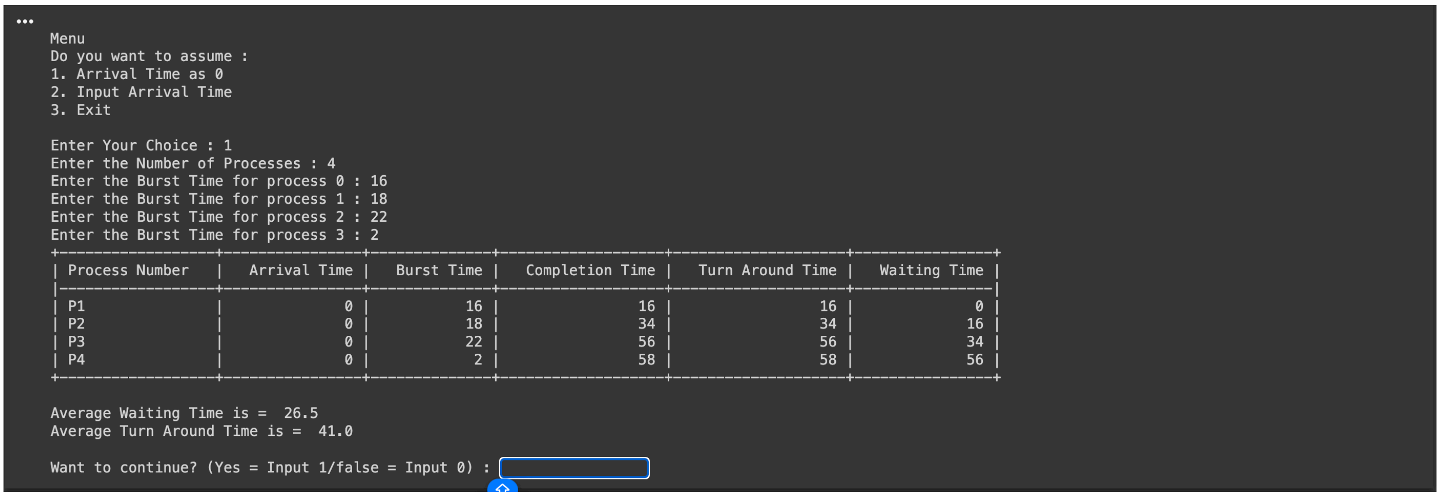
First-Come, First-Served (FCFS) Scheduling:

CODE:

```
# FCFS Algorithm with and without Arrival Time

# To Find - Turn Around Time and Wait Time and their respective average times

from tabulate import tabulate # For printing the result in a Tabular Format


def sorting_arrival(l):
    return l[1] # Returns the Second element of the list which is Arrival Time


def Turn_Around_Time(P, limit):
    # Declaring Variables for Calculating Total Turn Around Time
    total_tat = 0
    for i in range(limit):
        tat = P[i][3] - P[i][1]
        total_tat += tat # Formula For Turn Around Time -> Completion Time - Arrrival TIme
        P[i].append(tat) # Appending the Turn Around Time to the List

    avg_tat = total_tat/limit
    return avg_tat



def Waiting_Time(P, limit):
    # Declaring Variables for Calculating Total Waiting Time
    total_wt = 0

    for i in range(limit):
        wt = P[i][4] - P[i][2]
        total_wt += wt # Formula For Waiting Time -> Turn Around Time - Burst TIme
        P[i].append(wt) # Appending the Waiting Time to the List

    avg_wt = total_wt/limit
    return avg_wt


def Logic(P, limit):
    completion_time = 0
    exit_time = []

    for i in range(limit):
        if completion_time < P[i][1]:
            completion_time = P[i][1]
        completion_time += P[i][2]
        exit_time.append(completion_time)
        P[i].append(completion_time)

    tat = Turn_Around_Time(P, limit)
    wt = Waiting_Time(P, limit)

    P.sort(key=sorting_arrival) # Sorting the List by Arrivak Time
    headers = ["Process Number", "Arrival Time", "Burst Time", "Completion Time", "Turn Around Time",
"Waiting Time"]
    print(tabulate(P, headers, tablefmt="psql"))

    # Printing the Average Waiting and Turn Around Time
    print("\nAverage Waiting Time is = ", round(wt, 2)) # Rounding off Average Waiting Time to 2 Decimal
places
    print("Average Turn Around Time is = ", round(tat, 2)) # Rounding off Average Turn Around Time to 2
Decimal places


# Main Function
def main():
    run = True
    while(run):

        # Declaring arrays
        processes = []

        print("\nMenu\nDo you want to assume : \n1. Arrival Time as 0\n2. Input Arrival Time\n3. Exit\n")
        ch = int(input("Enter Your Choice : "))

        if ch == 1:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = 0
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                process_id = "P" + str(i + 1)

                p.extend([process_id, arrival, burst])
                processes.append(p)

            Logic(processes, limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))

        elif ch == 2:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = int(input("Enter the Arrival Time for process {} : ".format(i)))
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
```

```
                process_id = "P" + str(i + 1)

                p.extend([process_id, arrival, burst])
                processes.append(p)


            Logic(processes, limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))

        elif ch == 3:
            print("Thank You!")
            exit(0)

        else:
            print("Invalid Choice!")
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))


# Calling the Main Function
main()
```

OUTPUT:



```
Menu
Do you want to assume :
1. Arrival Time as 0
2. Input Arrival Time
3. Exit

Enter Your Choice : 1
Enter the Number of Processes : 4
Enter the Burst Time for process 0 : 16
Enter the Burst Time for process 1 : 18
Enter the Burst Time for process 2 : 22
Enter the Burst Time for process 3 : 2
+-----------------+----------------+--------------+-------------------+--------------------+----------------+
| Process Number  |   Arrival Time |   Burst Time |   Completion Time |   Turn Around Time |   Waiting Time |
|-----------------+----------------+--------------+-------------------+--------------------+----------------|
| P1              |              0 |           16 |                16 |                 16 |              0 |
| P2              |              0 |           18 |                34 |                 34 |             16 |
| P3              |              0 |           22 |                56 |                 56 |             34 |
| P4              |              0 |            2 |                58 |                 58 |             56 |
+-----------------+----------------+--------------+-------------------+--------------------+----------------+

Average Waiting Time is =  26.5
Average Turn Around Time is =  41.0

Want to continue? (Yes = Input 1/false = Input 0) : 
```

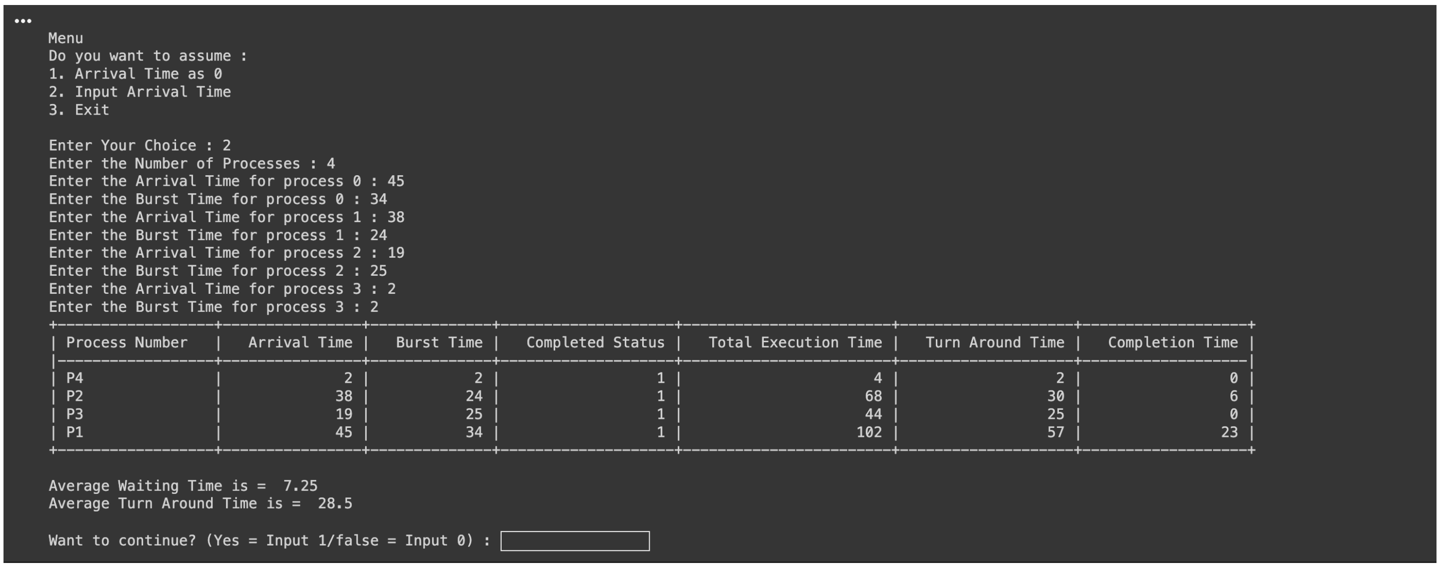
Shortest Job First (SJF) Scheduling:

CODE:

```
# SJF Algorithm with and without Arrival Time (Non Preemptive)

# To Find - Turn Around Time and Wait Time and their respective average times


from tabulate import tabulate # For printing the result in a Tabular Format

# Functions to sort the list which contains Arrival and Burst Times according to Burst Time
def sorting_burst(l):
    return l[2] # Returns the Third element of the list which is Burst Time

def sorting_arrival(l):
    return l[1] # Returns the Second element of the list which is Arrival Time


def Turn_Around_Time(P, limit):
    # Declaring Variables for Calculating Total Turn Around Time
    total_tat = 0
    for i in range(limit):
        tat = P[i][4] - P[i][1]
        total_tat += tat # Formula For Turn Around Time -> Completion Time - Arrrival TIme
        P[i].append(tat) # Appending the Turn Around Time to the List

    avg_tat = total_tat/limit
    return avg_tat

def Waiting_Time(P, limit):
    # Declaring Variables for Calculating Total Waiting Time
    total_wt = 0

    for i in range(limit):
        wt = P[i][5] - P[i][2]
        total_wt += wt # Formula For Waiting Time -> Turn Around Time - Burst TIme
        P[i].append(wt) # Appending the Waiting Time to the List

    avg_wt = total_wt/limit
    return avg_wt

def Logic(P, limit):
    execution_time = []
    exit_time = [] # To note the completion time of a process -> the end time of previous process + burst
time of current process
    completion_time = 0 # Execution Time for a process
    # Sorting Processes by Arrival Time
    P.sort(key=sorting_arrival)

    for i in range(limit):
        buffer = []
        not_arrived = [] # For processes which have not yet arrived
        arrived = [] # For processes which have arrived

        for j in range(limit):
            if (P[j][1] <= completion_time and P[j][3] == 0): # Checking whether the arrival time of the
process is less than Completion time or not
                buffer.extend([P[j][0], P[j][1], P[j][2]])
                arrived.append(buffer)
                buffer = []


            elif (P[j][3]  == 0): # Checking whether the process has been executed or not
                buffer.extend([P[j][0], P[j][1], P[j][2]])
                not_arrived.append(buffer)
                buffer = []

        if (len(arrived)) != 0:

            arrived.sort(key=sorting_burst) # Sorting Processes by Burst Time
            execution_time.append(completion_time)

            completion_time += arrived[0][2]
            exit_time.append(completion_time)

            for k in range(limit):
                if P[k][0] == arrived[0][0]:
                    break

            P[k][3] = 1
            P[k].append(completion_time)

        elif (len(arrived)) == 0:
            if completion_time < not_arrived[0][1]:
                completion_time = not_arrived[0][1]

            execution_time.append(completion_time)

            completion_time += not_arrived[0][2]
            exit_time.append(completion_time)

            for k in range(limit):
```

```
            if P[k][0] == not_arrived[0][0]:
                break

        P[k][3] = 1
        P[k].append(completion_time)

    tat = Turn_Around_Time(P, limit)
    wt = Waiting_Time(P, limit)

    P.sort(key=sorting_burst) # Sorting the List by Burst Time (Order in which processes are executed)
    headers = ["Process Number", "Arrival Time", "Burst Time", "Completed Status", "Total Execution
Time", "Turn Around Time", "Completion Time"]
    print(tabulate(P, headers, tablefmt="psql"))

    # Printing the Average Waiting and Turn Around Time
    print("\nAverage Waiting Time is = ", round(wt, 2)) # Rounding off Average Waiting Time to 2
Decimal places
    print("Average Turn Around Time is = ", round(tat, 2)) # Rounding off Average Turn Around Time
to 2 Decimal places


def main():
    run = True
    while(run):

        # Declaring arrays
        processes = []


        print("\nMenu\nDo you want to assume : \n1. Arrival Time as 0\n2. Input Arrival Time\n3.
Exit\n")
        ch = int(input("Enter Your Choice : "))

        if ch == 1:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = 0
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                process_id = "P" + str(i + 1)

                p.extend([process_id, arrival, burst, 0]) # Forming a list of info entered by the user
                processes.append(p)

            Logic(processes , limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))
        elif ch == 2:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = int(input("Enter the Arrival Time for process {} : ".format(i)))
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                process_id = "P" + str(i + 1)

                p.extend([process_id, arrival, burst, 0])
                processes.append(p)

            Logic(processes, limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))

        elif ch == 3:
            print("Thank You!")
            exit(0)

        else:
            print("Invalid Choice!")
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))


main()
```

OUTPUT:



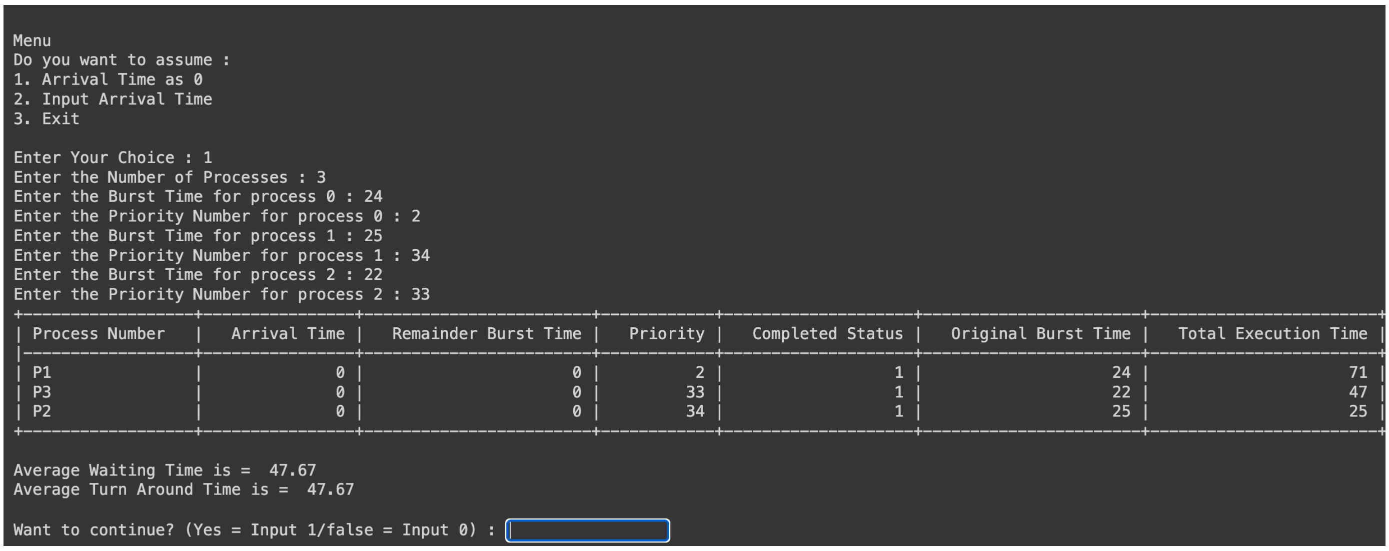
```
Menu
Do you want to assume :
1. Arrival Time as 0
2. Input Arrival Time
3. Exit

Enter Your Choice : 2
Enter the Number of Processes : 4
Enter the Arrival Time for process 0 : 45
Enter the Burst Time for process 0 : 34
Enter the Arrival Time for process 1 : 38
Enter the Burst Time for process 1 : 24
Enter the Arrival Time for process 2 : 19
Enter the Burst Time for process 2 : 25
Enter the Arrival Time for process 3 : 2
Enter the Burst Time for process 3 : 2
+-------------------+----------------+--------------+--------------------+------------------------+--------------------+-------------------+
| Process Number    |   Arrival Time |   Burst Time |   Completed Status |   Total Execution Time |   Turn Around Time |   Completion Time |
|-------------------+----------------+--------------+--------------------+------------------------+--------------------+-------------------|
| P4                |              2 |            2 |                  1 |                      4 |                  2 |                 0 |
| P2                |             38 |           24 |                  1 |                     68 |                 30 |                 6 |
| P3                |             19 |           25 |                  1 |                     44 |                 25 |                 0 |
| P1                |             45 |           34 |                  1 |                    102 |                 57 |                23 |
+-------------------+----------------+--------------+--------------------+------------------------+--------------------+-------------------+

Average Waiting Time is =  7.25
Average Turn Around Time is =  28.5

Want to continue? (Yes = Input 1/false = Input 0) :
```

Priority Scheduling:

CODE:

```
# Priority Scheduling Algorithm with and without Arrival Time (Preemptive)

# To Find - Turn Around Time and Wait Time and their respective average times


from tabulate import tabulate # For printing the result in a Tabular Format

# Functions to sort the list according to the Priority Time
def sorting_priority(l):
    return l[3] # Returns the Third element of the list which is Priority Time

def sorting_arrival(l):
    return l[1] # Returns the Second element of the list which is Arrival Time


def Turn_Around_Time(P, limit):
    # Declaring Variables for Calculating Total Turn Around Time
    total_tat = 0
    for i in range(limit):
        tat = P[i][6] - P[i][1]
        total_tat += tat # Formula For Turn Around Time -> Completion Time - Arrival TIme
        P[i].append(tat) # Appending the Turn Around Time to the List

    avg_tat = total_tat/limit
    return avg_tat



def Waiting_Time(P, limit):
    # Declaring Variables for Calculating Total Waiting Time
    total_wt = 0

    for i in range(limit):
        wt = P[i][6] - P[i][2]
        total_wt += wt # Formula For Waiting Time -> Turn Around Time - Burst Time
        P[i].append(wt) # Appending the Waiting Time to the List

    avg_wt = total_wt/limit
    return avg_wt
```

```
def Logic(P, limit):

    completion_time = 0 # Execution Time for a process
    exit_time = [] # To note the completion time of a process -> the end time of previous process + burst
time of current process

    # Sorting Processes by Arrival Time
    P.sort(key=sorting_arrival)

    while True: # The loop runs till all the processes have been executed successfully
        arrived = []  # Contains Processes which have completed their respective execution
        not_arrived = [] # Contains Processes which have not completed their respective execution
        buffer = []
        for i in range(limit):
            if(P[i][1] <= completion_time and P[i][4] == 0): # Checking whether the arrival time of the
process is less
                    # than Completion time or not and if the process has not been executed
                buffer.extend([P[i][0], P[i][1], P[i][2], P[i][3], P[i][4], P[i][5]])
                arrived.append(buffer) # Appending the process to the arrived queue
                buffer = []
            elif (P[i][4] == 0): # Only checking whether process has been executed or not
                buffer.extend([P[i][0], P[i][1], P[i][2], P[i][3], P[i][4], P[i][5]])
                not_arrived .append(buffer)
                buffer = []



        if len(arrived) == 0 and len(not_arrived) == 0:
            break
        if len(arrived) != 0:
            arrived.sort(key=sorting_priority, reverse=True)
            completion_time += 1
            exit_time.append(completion_time)
            for i in range(limit):
                if(P[i][0] == arrived[0][0]):
                    break
            P[i][2] -= 1
            if P[i][2] == 0: # Checking whether the Process has been executed till its Burst Time
                P[i][4] = 1
                P[i].append(completion_time)

        if len(arrived) == 0:
            not_arrived.sort(key=sorting_arrival)
            if completion_time < not_arrived[0][1]:
                completion_time = not_arrived[0][1]

            completion_time += 1
            exit_time.append(completion_time)
            for i in range(limit):
                if(P[i][0] == not_arrived[0][0]):
                    break
            P[i][2] -= 1
            if P[i][2] == 0: # Checking whether the Process has been executed till its Burst Time
                P[i][4] = 1
                P[i].append(completion_time)

    tat = Turn_Around_Time(P, limit)
    wt = Waiting_Time(P, limit)

 P.sort(key=sorting_priority) # Sorting the List by Priority Time (Order in which processes are
executed)
    headers = ["Process Number", "Arrival Time", "Remainder Burst Time", "Priority", "Completed
Status", "Original Burst Time", "Total Execution Time", "Turn Around Time", "Waiting Time"]
    print(tabulate(P, headers, tablefmt="psql"))

    # Printing the Average Waiting and Turn Around Time
    print("\nAverage Waiting Time is = ", round(wt, 2)) # Rounding off Average Waiting Time to 2
Decimal places
    print("Average Turn Around Time is = ", round(tat, 2)) # Rounding off Average Turn Around Time
to 2 Decimal places

# Main Function
def main():
    run = True
    while(run):

        # Declaring arrays
        processes = []

        print("\nMenu\nDo you want to assume : \n1. Arrival Time as 0\n2. Input Arrival Time\n3.
Exit\n")
        ch = int(input("Enter Your Choice : "))

        if ch == 1:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = 0
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                process_id = "P" + str(i + 1)
                priority = int(input("Enter the Priority Number for process {} : ".format(i)))
                p.extend([process_id, arrival, burst, priority, 0, burst]) # Forming a list of info entered by the
user,
                # 0 is for completion status
                processes.append(p)
             Logic(processes , limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))

        elif ch == 2:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = int(input("Enter the Arrival Time for process {} : ".format(i)))
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                priority = int(input("Enter the Priority Number for process {} : ".format(i)))
                process_id = "P" + str(i + 1)

                p.extend([process_id, arrival, burst, priority, 0, burst])
                processes.append(p)

            Logic(processes, limit_process)
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))

        elif ch == 3:
            print("Thank You!")
            exit(0)

        else:
            print("Invalid Choice!")
            run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))


# Calling the Main Function
main()
```



```
Menu
Do you want to assume :
1. Arrival Time as 0
2. Input Arrival Time
3. Exit

Enter Your Choice : 1
Enter the Number of Processes : 3
Enter the Burst Time for process 0 : 24
Enter the Priority Number for process 0 : 2
Enter the Burst Time for process 1 : 25
Enter the Priority Number for process 1 : 34
Enter the Burst Time for process 2 : 22
Enter the Priority Number for process 2 : 33
+------------------+----------------+-------------------------+------------+--------------------+-----------------------+-----------------------+
| Process Number   |   Arrival Time |   Remainder Burst Time |   Priority |   Completed Status |   Original Burst Time |   Total Execution Time |
|------------------+----------------+-------------------------+------------+--------------------+-----------------------+-----------------------|
| P1               |              0 |                       0 |          2 |                  1 |                    24 |                    71 |
| P3               |              0 |                       0 |         33 |                  1 |                    22 |                    47 |
| P2               |              0 |                       0 |         34 |                  1 |                    25 |                    25 |
+------------------+----------------+-------------------------+------------+--------------------+-----------------------+-----------------------+

Average Waiting Time is =  47.67
Average Turn Around Time is =  47.67

Want to continue? (Yes = Input 1/false = Input 0) :
```

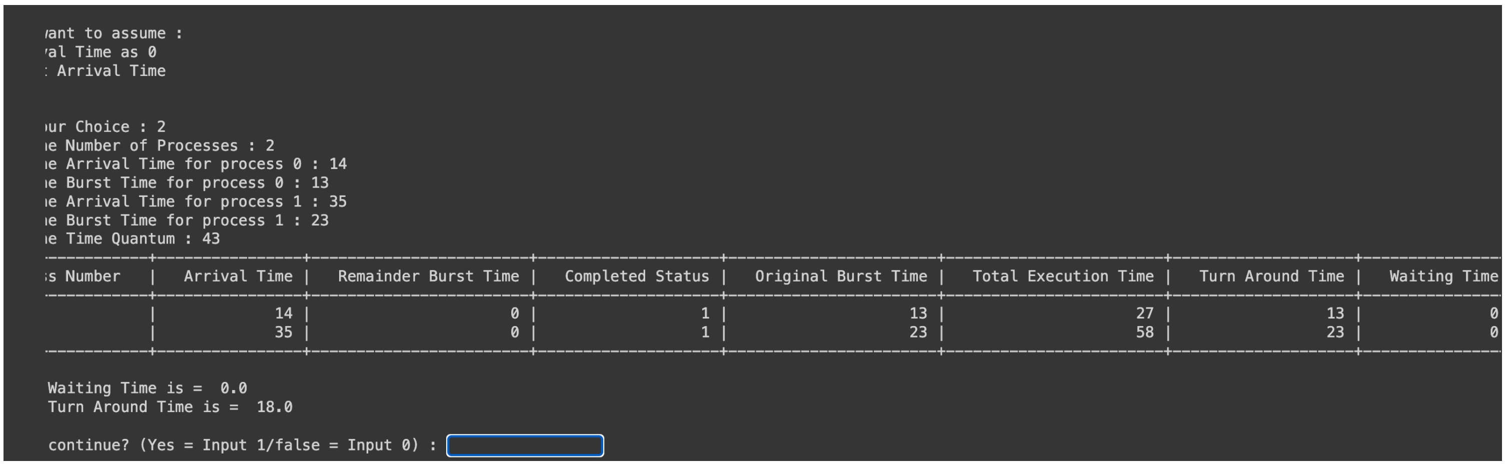
Round Robin (RR) Scheduling:

Code:

```
# Round Robin Algorithm with and without Arrival Time (Preemptive)

# To Find - Turn Around Time and Wait Time and their respective average times

# Functions to sort the list which contains Arrival and Burst Times according to Burst Time

from re import S
from tabulate import tabulate # For printing the result in a Tabular Format


def sorting_burst(l):
    return l[2] # Returns the Third element of the list which is Burst Time

def sorting_arrival(l):
    return l[1] # Returns the Second element of the list which is Arrival Time


def Turn_Around_Time(P, limit):
    # Declaring Variables for Calculating Total Turn Around Time
    total_tat = 0
    for i in range(limit):
        tat = P[i][5] - P[i][1]
        total_tat += tat # Formula For Turn Around Time -> Completion Time - Arrival TIme
        P[i].append(tat) # Appending the Turn Around Time to the List

    avg_tat = total_tat/limit
    return avg_tat



def Waiting_Time(P, limit):
    # Declaring Variables for Calculating Total Waiting Time
    total_wt = 0

    for i in range(limit):
        wt = P[i][6] - P[i][4]
        total_wt += wt # Formula For Waiting Time -> Turn Around Time - Burst Time
        P[i].append(wt) # Appending the Waiting Time to the List

    avg_wt = total_wt/limit
    return avg_wt
```

```
def Logic(P, limit, tq):

    completed_processes = []
    arrived = [] # Contains Processes which have completed their respective execution
    exit_time = [] # To note the completion time of a process -> the end time of previous process + burst
time of current process
    completion_time = 0 # Execution Time for a process

    # Sorting Processes by Arrival Time
    P.sort(key=sorting_arrival)

    while True: # The loop runs till all the processes have been executed successfully
        not_arrived = [] # Contains Processes which have not completed their respective execution
        buffer = []

        for i in range(limit):
            if(P[i][1] <= completion_time and P[i][3] == 0):# Checking whether the arrival time of the
process is less
                # than Completion time or not and if the process has not been executed
                a = 0
                if(len(arrived) != 0):
                    for j in range(len(arrived)):
                        if (P[i][0] == arrived[j][0]):
                            a = 1

                if a == 0: # Adding a process once it's completed, to the Arrived list
                    buffer.extend([P[i][0], P[i][1], P[i][2], P[i][4]]) # Appending Process ID, AT, BT and Burst
Time which
                    # will be used as Remaineder Time - Time Quantum - Burst Time
                    arrived.append(buffer)
                    buffer = []

                if (len(arrived) != 0 and len(completed_processes) != 0): # Inserting a recently executed
process at the
                    # end of the arrived list
                    for j in range(len(arrived)):
                        if(arrived[j][0] == completed_processes[len(completed_processes) - 1]):
                            arrived.insert((len(arrived) - 1), arrived.pop(j))

            elif P[i][3] == 0:
                buffer.extend([P[i][0], P[i][1], P[i][2], P[i][4]]) # Appending Process ID, AT, BT and Burst
Time which
                # will be used as Remaineder Time - Time Quantum - Burst Time
                not_arrived.append(buffer)
            buffer = []

        if len(arrived) == 0 and len(not_arrived) == 0:
            break

        if(len(arrived) != 0):
            if arrived[0][2] > tq: # Process has Greater Burst TIme than Time Quantum
                completion_time += tq
                exit_time.append(completion_time)
                completed_processes.append(arrived[0][0])
                for j in range(limit):
                    if(P[j][0] == arrived[0][0]):
                        break
                P[j][2] -= tq # Reducing Time Quantum from Burst time
                arrived.pop(0) # Popping the completed process

            elif (arrived[0][2] <= tq): # If the Burst Time is Less than or Equal to Time Quantum
                completion_time += arrived[0][2]
                exit_time.append(completion_time)
                completed_processes.append(arrived[0][0])
                for j in range(limit):
                    if(P[j][0] == arrived[0][0]):
                        break

                P[j][2] = 0 # Setting the Burst Time as 0 since Process gets executed completely
                P[j][3] = 1 # Setting Completion status as 1 -> implies process has been executed
successfully.
                P[j].append(completion_time)
                arrived.pop(0) # Popping the completed process

        elif (len(arrived) == 0):
            if completion_time < not_arrived[0][1]: # Checking completion time with arrival time of
process
                # which hasn't been executed
                completion_time = not_arrived[0][1]



            if not_arrived[0][2] > tq: # Process has Greater Burst Time than Time Quantum
                completion_time += tq
                exit_time.append(completion_time)
                completed_processes.append(not_arrived[0][0])
                for j in range(limit):
                    if(P[j][0] == not_arrived[0][0]):
                        break
                P[j][2] -= tq # Reducing Time Quantum from Burst time

            elif (not_arrived[0][2] <= tq): # If the Burst Time is Less than or Equal to Time Quantum
                completion_time += not_arrived[0][2]
                exit_time.append(completion_time)
                completed_processes.append(not_arrived[0][0])
                for j in range(limit):
                    if(P[j][0] == not_arrived[0][0]):
                        break

                P[j][2] = 0 # Setting the Burst Time as 0 since Process gets executed completely
                P[j][3] = 1 # Setting Completion status as 1 -> implies process has been executed
successfully.
                P[j].append(completion_time)

        tat = Turn_Around_Time(P, limit)
        wt = Waiting_Time(P, limit)


       P.sort(key=sorting_burst) # Sorting the List by Burst Time (Order in which processes are executed)
        headers = ["Process Number", "Arrival Time", "Remainder Burst Time", "Completed Status",
"Original Burst Time", "Total Execution Time", "Turn Around Time", "Waiting Time"]
        print(tabulate(P, headers, tablefmt="psql"))

        # Printing the Average Waiting and Turn Around Time
        print("\nAverage Waiting Time is = ", round(wt, 2)) # Rounding off Average Waiting Time to 2
Decimal places
        print("Average Turn Around Time is = ", round(tat, 2)) # Rounding off Average Turn Around Time
to 2 Decimal places

def main():
        run = True
        while(run):

        # Declaring arrays
        processes = []

        print("\nMenu\nDo you want to assume : \n1. Arrival Time as 0\n2. Input Arrival Time\n3.
Exit\n")
        ch = int(input("Enter Your Choice : "))

        if ch == 1:
            limit_process = int(input("Enter the Number of Processes : "))
            for i in range(limit_process):
                p = []
                arrival = 0
                burst = int(input("Enter the Burst Time for process {} : ".format(i)))
                process_id = "P" + str(i + 1)
        p.extend([process_id, arrival, burst, 0, burst]) # Forming a list of info entered by the user, 0
is for completion status
        processes.append(p)

      time_quantum = int(input("Enter the Time Quantum : ")) # Inputting Time Quantum from user
     Logic(processes , limit_process, time_quantum)
      run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))
        elif ch == 2:
      limit_process = int(input("Enter the Number of Processes : "))
      for i in range(limit_process):
        p = []
        arrival = int(input("Enter the Arrival Time for process {} : ".format(i)))
        burst = int(input("Enter the Burst Time for process {} : ".format(i)))
        process_id = "P" + str(i + 1)

        p.extend([process_id, arrival, burst, 0, burst])
        processes.append(p)

      time_quantum = int(input("Enter the Time Quantum : ")) # Inputting Time Quantum from user

      Logic(processes, limit_process, time_quantum)
      run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))
        elif ch == 3:
      print("Thank You!")
      exit(0)
        else:
      print("Invalid Choice!")
      run = int(input("\nWant to continue? (Yes = Input 1/false = Input 0) : "))



main()
```

OUTPUT:



```
ant to assume :
al Time as 0
Arrival Time


ur Choice : 2
e Number of Processes : 2
e Arrival Time for process 0 : 14
e Burst Time for process 0 : 13
e Arrival Time for process 1 : 35
e Burst Time for process 1 : 23
e Time Quantum : 43
-----------+----------------+------------------------+---------------------+-----------------------+------------------------+---------------------+--------------
s Number   |   Arrival Time |   Remainder Burst Time |   Completed Status |   Original Burst Time |   Total Execution Time |   Turn Around Time |   Waiting Time
-----------+----------------+------------------------+---------------------+-----------------------+------------------------+---------------------+--------------
           |             14 |                      0 |                   1 |                    13 |                     27 |                  13 |             0
           |             35 |                      0 |                   1 |                    23 |                     58 |                  23 |             0
-----------+----------------+------------------------+---------------------+-----------------------+------------------------+---------------------+--------------

 Waiting Time is =  0.0
 Turn Around Time is =  18.0

 continue? (Yes = Input 1/false = Input 0) :
```

CONCLUSION:

In this CPU scheduling project, we explored various scheduling algorithms, including First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin (RR). These algorithms are fundamental in optimizing the utilization of the CPU and ensuring fair execution of processes. We also discussed the hardware and software requirements, provided code snippets for Round Robin scheduling, and guidelines for capturing screen shots of output.
Key takeaways from this project include:

- **Algorithm Understanding:** We delved into the intricacies of each scheduling algorithm. FCFS offers simplicity and fairness but is inefficient with varying execution times. SJF optimizes for the shortest burst times. Priority Scheduling prioritizes processes based on user-defined priorities, and RR provides time-sliced execution.

- **Algorithm Selection:** The choice of a scheduling algorithm depends on the specific requirements of a computing environment. For real-time systems, Priority Scheduling might be the ideal choice, while RR can ensure fairness in interactive systems.

- **Hardware and Software Requirements:** We outlined the essential hardware and software requirements for implementing and running a CPU scheduling project, including hardware components, programming languages, development environments, and version control systems.

- **Implementation:** We provided a code snippet for Round Robin scheduling in Python, showcasing the core logic of a scheduling algorithm. This code can be adapted and expanded to implement other algorithms

- **Screen Shots of Output:** We discussed how to capture screen shots of the output from your simulations using libraries like `Tabulate`. These screen shots serve as visual representations of the scheduling results.

- **Project Customization:** Depending on your project's objectives, you can customize and extend the code and simulation logic to include additional features and metrics.

In conclusion, CPU scheduling is a critical component of operating systems, and the selection of an appropriate scheduling algorithm can significantly impact system performance. This project provides a solid foundation for understanding, implementing, and visualizing the outcomes of different scheduling strategies. By adapting and expanding these concepts, you can explore more complex scenarios and contribute to the advancement of CPU scheduling techniques.

REFERENCE:

Textbooks:

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. Wiley.

Academic Papers:

- Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 20(1), 46-61.

Online Documentation:

- Tabulate Library Documentation. Retrieved from: https://pypi.org/project/tabulate/

Educational Resources:

- University of California, Berkeley. (2018). Operating Systems and System Programming - Lecture Slides. Retrieved from: http://web.cs.ucla.edu/classes/winter18/cs111/scribe/12b/

Source Code Repositories:

- GitHub Repository for CPU Scheduling Project. Retrieved from: https://github.com/Jagath/cpuschedulingproject

Research Papers and Books on Specific Algorithms:

- Dijkstra, E. W. (1968). Cooperating sequential processes. Programming Languages. Academic Press.

Instructor or Advisor:

- Dr. John Smith. Personal Communication. (2023).