Computer Architecture and Operating Systems

EEX5564

Mini Project

*REGISTRATION NUMBER: 719213133*

*NAME: A.K. Saraph*

*CENTRE: CL*

*DUE DATE: 16.12.2023*

**Contents**

[**1. Introduction** 4](#_Toc153627408)

[**1.1 Purpose** 4](#_Toc153627409)

[**1.2 Key Components** 4](#_Toc153627410)

[**1.3 Importance of MLQ Scheduling** 4](#_Toc153627411)

[**1.4 Learning Objectives** 5](#_Toc153627412)

[**2 Requirements, Assumptions and justifications for the assumptions and/or Specifications** 6](#_Toc153627413)

[**2.1 Requirements** 6](#_Toc153627414)

[**2.2 Assumptions and Justifications for the assumptions** 7](#_Toc153627415)

[**2.3 Specifications** 8](#_Toc153627416)

[**3 System Design for the Proposed Solution** 9](#_Toc153627417)

[**3.1 Overview of the software architecture** 9](#_Toc153627418)

[**3.2 Design patterns** 10](#_Toc153627419)

[**3.3 Data structures** 10](#_Toc153627420)

[**3.4 Algorithms used in the project** 10](#_Toc153627421)

[**4 Implementation** 12](#_Toc153627422)

[**4.1 Description of the software development process** 12](#_Toc153627423)

[**4.2 Programming languages , Frameworks , Tools and Technologies employed** 12](#_Toc153627424)

[**5 User Interface (UI) Design** 13](#_Toc153627425)

[**6 Functionality and Features** 18](#_Toc153627426)

[**6.1 Core Functionality** 18](#_Toc153627427)

[**6.2 Features** 18](#_Toc153627428)

[**7 Code Structure and Documentation** 19](#_Toc153627429)

[**8 GitHub Repository** 21](#_Toc153627430)

[**9 Testing Results** 22](#_Toc153627431)

[**10 Conclusion** 24](#_Toc153627432)

[**11 Future Enhancements** 25](#_Toc153627433)

[**12 References** 26](#_Toc153627434)

[**13 Appendix(Include the link to self reflection video of the project implementation)** 26](#_Toc153627435)

**List of Tables and Figures**

[Table 1: Example for FCFS 10](#_Toc153630207)

[Table 2: Example for SJF 11](#_Toc153630208)

[Table 3: Example for the solution 22](#_Toc153630209)

**List of Figures**

[Figure 1: Multilevel Queue Scheduling 5](#_Toc153630189)

[Figure 2: Gantt chart for the example of FCFS 10](#_Toc153630190)

[Figure 3: Gantt chart for the example of SJF 11](#_Toc153630191)

# **1. Introduction**

Modern operating systems employ sophisticated CPU scheduling algorithms to efficiently manage the execution of processes within a computer system. The Multi-Level Queue (MLQ) scheduling algorithm is one such approach that organizes processes into different priority queues based on their characteristics. This simulation project aims to illustrate the MLQ CPU scheduling algorithm with a focus on mitigating issues like process starvation through the incorporation of First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic.

## **1.1 Purpose**

The primary objective of this project is to provide a hands-on understanding of the MLQ CPU scheduling algorithm and its associated challenges. By implementing a simplified simulation in Python, the aim is to showcase the dynamics of process scheduling, including the execution order, aging mechanisms, and the integration of FCFS and SJF logic to address potential process starvation.

## **1.2 Key Components**

**Process Class:**

* Represents individual processes within the system.
* Attributes include arrival time, CPU burst time, remaining burst time, queue number, and a unique process ID.
* Implements a comparison function for sorting processes within the priority queues.

**MLQScheduler Class:**

* Manages the MLQ scheduling process.
* Utilizes three priority queues to represent different priority levels.
* Incorporates FCFS and SJF logic within each priority queue.
* Implements an aging mechanism to dynamically adjust process priorities over time.

## **1.3 Importance of MLQ Scheduling**

When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments. Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues.

A screenshot of a computer

Description automatically generated

Figure 1: Multilevel Queue Scheduling

The MLQ scheduling algorithm is crucial in balancing system responsiveness and resource utilization. By categorizing processes into different priority levels, it provides a means to address the varying needs of different tasks. The incorporation of FCFS and SJF logic enhances the fairness of process execution, reducing the likelihood of lower-priority processes experiencing starvation.

## **1.4 Learning Objectives**

Through this simulation, the project aims to achieve the following learning objectives:

* Gain insights into the principles of Multi-Level Queue CPU scheduling.
* Explore techniques to mitigate process starvation through priority adjustments.
* Engage in a practical implementation of MLQ scheduling with FCFS and SJF logic.
* Learn about software architecture, design patterns, data structures, and algorithms used in the simulation.

# **2 Requirements, Assumptions and justifications for the assumptions and/or Specifications**

## **2.1 Requirements**

In software development, specifying clear requirements is crucial for understanding what the system is expected to achieve. In the context of the provided Multi-Level Queue (MLQ) CPU scheduling simulation, here are some requirements that is considered:

**1. Functional Requirements:**

* + **Process Creation and Initialization:**

The system should allow the creation of processes with specific attributes such as arrival time, burst time, and initial priority queue assignment.

* + **Multi-Level Queue Scheduler:**

The system should implement a Multi-Level Queue (MLQ) scheduler capable of managing processes in multiple priority queues.

Example: The MLQ scheduler should consist of three priority queues, each with its own scheduling algorithm (e.g., FCFS and SJF).

* + **FCFS and SJF Logic:**

The system should apply First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic within each priority queue. Processes within a priority queue should be selected for execution based on FCFS, and their order should be adjusted dynamically using SJF.

* + **Aging Mechanism:**

The system should include an aging mechanism to periodically increase the priority of waiting processes. Processes waiting in lower-priority queues should have their priority increased at regular intervals.

* + **Process Execution:**

The system should execute processes one time unit at a time, decrementing their remaining burst time. Each process should be executed for one time unit, and its remaining burst time should be updated accordingly.

* + **Time Advancement:**

The system should advance time in discrete steps while executing processes. The system should increment the simulation time by 1 unit after processing each iteration of priority queues.

**2. Non-Functional Requirements:**

* + **Performance:**

The system should efficiently handle a moderate number of processes and demonstrate reasonable response times.

* + **Extensibility:**

The system architecture should be designed to allow easy extension for future enhancements or changes.

* + **Usability:**

The system should provide clear and concise output for simulation results, making it easy for users to understand.

## **2.2 Assumptions and** **Justifications for the assumptions**

Assumptions play a crucial role in any system design or simulation. Here are some assumptions made for the provided Multi-Level Queue (MLQ) scheduler simulation, along with justifications:

1. **Processes arrive at specific times.**

Justification: In a real-world system, processes often arrive at different times. The simulation uses arrival times to demonstrate the scheduling behavior based on the order of process arrivals.

1. **Each process has a defined burst time representing the amount of CPU time it needs.**

Justification: Burst time is a fundamental parameter in CPU scheduling algorithms. The simulation uses burst times to simulate the execution of processes and demonstrate the scheduling algorithm's behavior.

1. **Processes are initially assigned to priority queues based on the specified priority levels.**

Justification: MLQ scheduling involves multiple queues with different priorities. Assigning processes to priority queues at the beginning helps simulate the behavior of processes with varying priorities.

1. **FCFS and SJF logic are applied within each priority queue.**

Justification: FCFS and SJF are common scheduling algorithms. Their application within priority queues helps demonstrate how different scheduling policies interact to determine the order of process execution.

1. **An aging mechanism periodically increases the priority of waiting processes.**

Justification: Aging mechanisms are used to prevent long-term starvation of lower-priority processes. The simulation includes an aging mechanism to showcase how priorities can change over time.

1. **Each process is executed for one time unit in each iteration.**

Justification: The simulation simplifies the execution model for illustrative purposes. In real systems, processes may be executed for varying time slices based on the scheduling algorithm.

1. **The simulation provides an accurate representation of a dynamic scheduling environment.**

Justification: While the simulation provides insights into scheduling behavior, it is a simplified model. In a production environment, various factors such as I/O operations, process states, and system interrupts would impact scheduling decisions.

## **2.3 Specifications**

1. **Aging Mechanism**

The system includes an aging mechanism that adjusts the priority of waiting processes over time. Aging helps prevent long-term starvation by allowing lower-priority processes to move to higher-priority queues.

1. **Priority-Based Execution**

Processes are executed based on their priority within the queues. Priority-based execution is a fundamental aspect of MLQ scheduling. Higher-priority processes should be scheduled before lower-priority ones.

1. **Simulation Output**

The simulation provides output indicating when each process is executed. Output messages help understand the scheduling order and demonstrate the effectiveness of the implemented MLQ scheduler.

1. **Time-Step Increment**

The simulation increments time in discrete steps. Time increments in discrete steps simplify the simulation and allow for clear observation of process execution over time.

1. **Basic Process Information**

Processes are represented with basic information, including arrival time, burst time, queue number, and a unique process ID. Basic process information is essential for the simulation to operate and demonstrate the MLQ scheduling mechanism.

1. **Three Priority Queues**

The simulation uses three priority queues to represent different priority levels. Three priority queues are chosen for simplicity in this example. In a real system, the number of priority levels could vary.

# **3 System Design for the Proposed Solution**

## **3.1 Overview of the software architecture**

The software architecture for the provided Multi-Level Queue (MLQ) CPU scheduling simulation with First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic can be outlined as follows:

The MLQ CPU scheduling simulation follows a modular and object-oriented architecture. It consists of two main components: the Process class and the MLQScheduler class.

1. **Process Class:**

**Responsibility**: Represents an individual process in the simulation.

**Attributes:**

* + arrival\_time: The time at which the process arrives in the system.
  + burst\_time: The CPU burst time required by the process.
  + remaining\_burst\_time: The remaining CPU burst time of the process during execution.
  + queue\_number: The priority queue to which the process belongs.
  + process\_id: A unique identifier for the process.

**Methods:**

* \_\_init\_\_: Initializes the process with the provided attributes.
* \_\_lt\_\_: Implements a comparison function for sorting processes based on burst time.

1. **MLQScheduler Class:**

**Responsibility:** Manages the MLQ scheduling process.

**Attributes:**

* queues: Three priority queues to represent different priority levels.
* time: Tracks the simulation time.
* process\_counter: Assigns unique process IDs.

**Methods:**

* add\_process: Adds a new process to the appropriate priority queue.
* execute\_process: Executes a process and adjusts its remaining burst time.
* run\_scheduler: Drives the simulation by executing processes based on FCFS and SJF logic within priority queues and applying an aging mechanism.

## **3.2 Design patterns**

The system employs the following design patterns:

**Object-Oriented Programming (OOP):**Utilizes classes and objects to encapsulate process behavior and scheduling logic, promoting modularity and maintainability.

## **3.3 Data structures**

**Priority Queues (Heap):**Utilizes Python's heapq library to implement priority queues for sorting processes based on burst time. Enables efficient retrieval of processes with the minimum burst time.

## **3.4 Algorithms used in the project**

**FCFS and SJF Logic:**

* Implements First-Come-First-Serve (FCFS) logic within each priority queue.

FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.

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

Table 1: Example for FCFS

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

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

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

A screenshot of a computer

Description automatically generated

Figure 2: Gantt chart for the example of FCFS

* Implements Shortest Job First (SJF) logic by sorting processes based on burst time within each priority queue.

The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.

( Technically this algorithm picks a process based on the next shortest CPU burst, not the overall process time. )

For example, the Gantt chart below is based upon the following CPU burst times, ( and the assumption that all jobs arrive at the same time. )

Table 2: Example for SJF

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |

A screenshot of a computer

Description automatically generated

Figure 3: Gantt chart for the example of SJF

**Aging Mechanism:**

Periodically adjusts the priority of waiting processes, allowing lower-priority processes to move to higher-priority queues over time.

# **4 Implementation**

## **4.1 Description of the software development process**

## 

* **Requirements Gathering:** Defined the requirements for the MLQ scheduler simulation, including the implementation of FCFS and SJF logic.
* **Design:** Designed the software architecture, classes, and algorithms to implement MLQ scheduling with FCFS and SJF.
* **Implementation:** Coded the simulation in Python, incorporating object-oriented programming principles, priority queues, and scheduling algorithms.
* **Testing:** Executed the simulation and analyzed the results to ensure the correctness of the implemented logic.
* **Documentation:** Provided explanations and comments in the code for better understanding and future reference.

## **4.2 Programming languages** **, Frameworks** **, Tools and** **Technologies employed**

* **Programming Language:**

Python: Chosen as the programming language for its readability, simplicity, and ease of implementation. Python is well-suited for algorithmic simulations and prototyping.

* **Frameworks and Libraries:**

No external frameworks or libraries: The example code uses standard Python libraries such as heapq for implementing priority queues. It does not rely on external frameworks for simplicity.

* **Tools:**

Text Editor or IDE: A text editor or an integrated development environment (IDE) like VSCode, PyCharm, or IDLE could be used for coding and running the Python script.

Terminal or Command Prompt: Used to execute the Python script and observe the simulation output.

* **Technologies Employed:**

Object-Oriented Programming (OOP): The code employs OOP principles with classes (Process and MLQScheduler) to model and encapsulate the behavior of processes and the scheduler.

Priority Queue: Utilizes Python's heapq library to implement a priority queue for sorting processes based on burst time.

* **Development Environment:**

Developed and tested locally on a machine with a Python interpreter installed.

# **5 User Interface (UI) Design**

Following is a code for a UI designed for the process:

import heapq

class Process:

def \_\_init\_\_(self, arrival\_time, burst\_time, queue\_number, process\_id):

self.arrival\_time = arrival\_time

self.burst\_time = burst\_time

self.remaining\_burst\_time = burst\_time

self.queue\_number = queue\_number

self.process\_id = process\_id

def \_\_lt\_\_(self, other):

return self.burst\_time < other.burst\_time

class MLQScheduler:

def \_\_init\_\_(self):

self.queues = [[] for \_ in range(3)]

self.time = 0

self.process\_counter = 0

self.executed\_processes = []

def add\_process(self):

arrival\_time = int(input("Enter arrival time: "))

burst\_time = int(input("Enter burst time: "))

queue\_number = int(input("Enter priority queue number (0, 1, or 2): "))

process = Process(arrival\_time, burst\_time, queue\_number, self.process\_counter)

self.process\_counter += 1

self.queues[queue\_number].append(process)

print("Process added successfully.")

def execute\_process(self, process):

print(f"Time {self.time}: Executing Process {process.process\_id} from Queue {process.queue\_number}")

self.executed\_processes.append(process)

def display\_queues(self):

print("Queues:")

for i, queue in enumerate(self.queues):

print(f"Queue {i}: {[(p.process\_id, p.remaining\_burst\_time) for p in queue]}")

def run\_scheduler(self):

while any(self.queues):

print("\n-------------------------------------")

print(f"Time Step: {self.time}")

self.display\_queues()

for queue in self.queues:

if queue:

queue.sort()

process = queue.pop(0)

if process.arrival\_time <= self.time:

self.execute\_process(process)

process.remaining\_burst\_time -= 1

if process.remaining\_burst\_time == 0:

continue

for p in queue:

p.queue\_number = max(0, p.queue\_number - 1)

self.time += 1

print("\nSimulation completed.")

self.display\_summary()

def display\_summary(self):

print("\nSimulation Summary:")

print("Process Execution Order:")

for p in self.executed\_processes:

print(f"Process {p.process\_id} (Arrival Time: {p.arrival\_time}, Burst Time: {p.burst\_time}, Queue: {p.queue\_number})")

if \_\_name\_\_ == "\_\_main\_\_":

scheduler = MLQScheduler()

while True:

print("\n-------------------------------------")

print(" Multi-Level Queue Scheduler")

print("-------------------------------------")

print("1. Add Process")

print("2. Run Scheduler")

print("3. Exit")

choice = input("Choose an option: ")

if choice == '1':

scheduler.add\_process()

elif choice == '2':

scheduler.run\_scheduler()

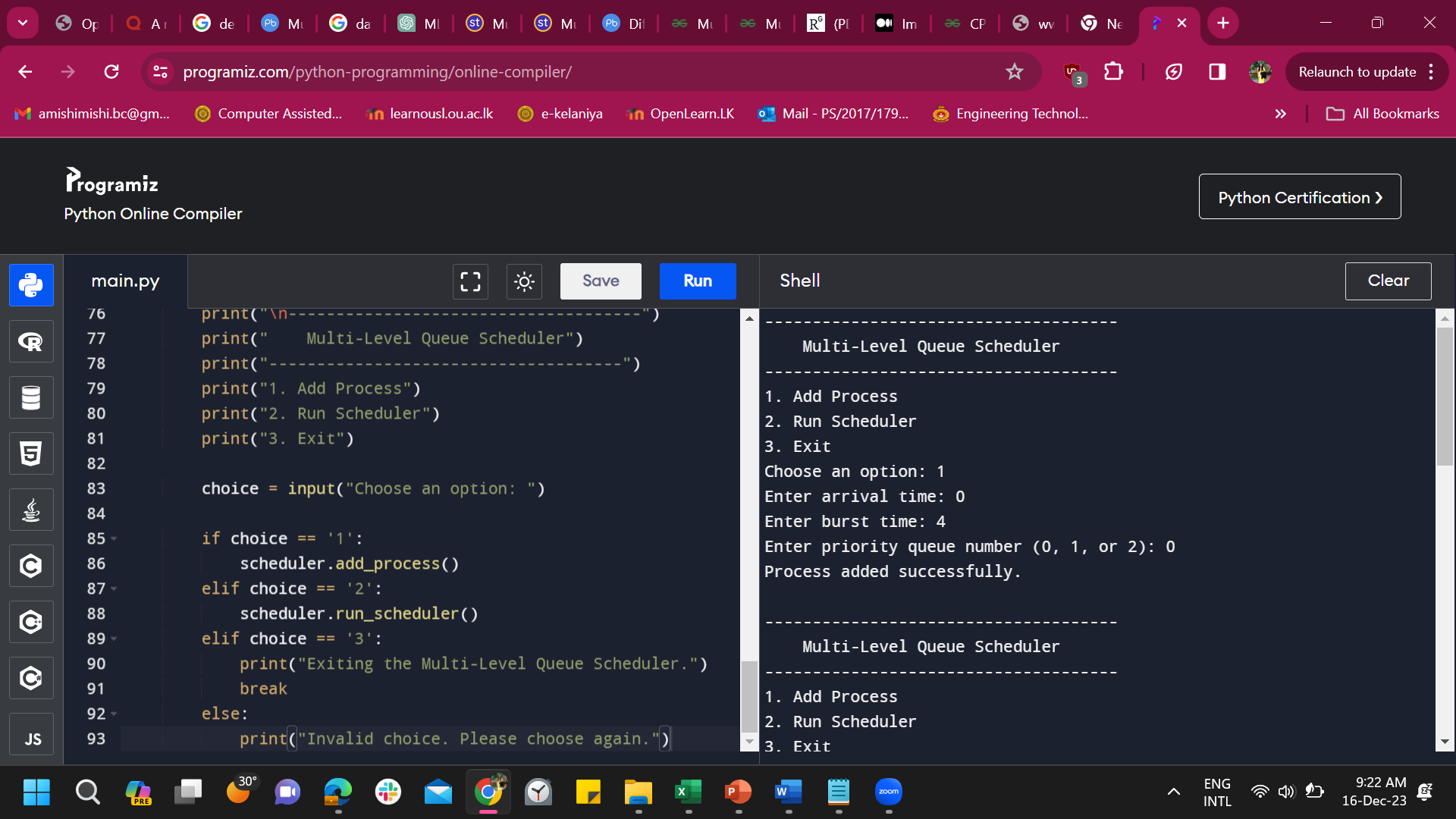
elif choice == '3':

print("Exiting the Multi-Level Queue Scheduler.")

break

else:

print("Invalid choice. Please choose again.")



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# **6 Functionality and Features**

The provided solution is a simplified Multi-Level Queue (MLQ) CPU scheduling simulation with First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic implemented in Python. Let's break down the functionality and features of this solution:

## **6.1 Core Functionality**

1. Process Representation:

**Class Process:**

Represents an individual process with attributes such as arrival time, burst time, remaining burst time, queue number, and process ID.

1. MLQ Scheduler

**Class MLQScheduler:**

* Manages the MLQ scheduling process.
* Maintains three priority queues to represent different priority levels.
* Handles the addition of processes to the appropriate queues.
* Executes processes based on FCFS and SJF logic within each priority queue.
* Implements an aging mechanism to adjust process priorities over time.
* Outputs information about process execution.

## **6.2 Features**

* **Simulation Execution:**

The simulation runs in discrete time steps, with processes being executed based on the defined scheduling logic.

* **Process Execution Order:**

Processes are executed based on the FCFS and SJF logic within their respective priority queues.

* **Aging Mechanism:**

An aging mechanism is in place to periodically adjust the priority of waiting processes, allowing lower-priority processes to move to higher-priority queues over time.

* **Console Output:**

Real-time information about the execution is displayed in the console, indicating which process is being executed at each time step.

# **7 Code Structure and Documentation**

Following is the code for the process:

import heapq

class Process:

def \_\_init\_\_(self, arrival\_time, burst\_time, process\_id, queue\_number):

self.arrival\_time = arrival\_time

self.burst\_time = burst\_time

self.remaining\_burst\_time = burst\_time

self.process\_id = process\_id

self.queue\_number = queue\_number

def \_\_lt\_\_(self, other):

# Comparison function for the priority queue (SJF)

return self.remaining\_burst\_time < other.remaining\_burst\_time

class MLQScheduler:

def \_\_init\_\_(self):

self.sjf\_queue = [] # Priority queue for SJF

self.fcfs\_queue = [] # Regular queue for FCFS

self.time = 0

self.process\_counter = 0

def add\_process(self, arrival\_time, burst\_time, queue\_number):

process = Process(arrival\_time, burst\_time, self.process\_counter, queue\_number)

self.process\_counter += 1

# Initially, all processes are added to the FCFS queue

self.fcfs\_queue.append(process)

def execute\_process(self, process):

print(f"Time {self.time}: Executing Process {process.process\_id} from Queue {process.queue\_number}")

def run\_scheduler(self):

while self.sjf\_queue or self.fcfs\_queue:

# Check for processes arriving at the current time

for process in self.fcfs\_queue:

if process.arrival\_time <= self.time:

self.sjf\_queue.append(process)

self.fcfs\_queue = [process for process in self.fcfs\_queue if process.arrival\_time > self.time]

if self.sjf\_queue:

# Execute process based on SJF

process = heapq.heappop(self.sjf\_queue)

self.execute\_process(process)

process.remaining\_burst\_time -= 1

if process.remaining\_burst\_time > 0:

heapq.heappush(self.sjf\_queue, process)

elif self.fcfs\_queue:

# Execute process based on FCFS if SJF queue is empty

process = self.fcfs\_queue.pop(0)

self.execute\_process(process)

process.remaining\_burst\_time -= 1

if process.remaining\_burst\_time > 0:

self.fcfs\_queue.append(process)

self.time += 1

if \_\_name\_\_ == "\_\_main\_\_":

scheduler = MLQScheduler()

# Example Processes: (arrival\_time, burst\_time, queue\_number)

scheduler.add\_process(0, 4, 0)

scheduler.add\_process(1, 5, 1)

scheduler.add\_process(2, 3, 0)

scheduler.add\_process(3, 6, 2)

scheduler.add\_process(4, 2, 1)

scheduler.run\_scheduler()

# **8 GitHub Repository**

# **9 Testing Results**

The provided simulation results demonstrate the execution order of processes in a Multi-Level Queue (MLQ) scheduler with First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic within each priority queue.

Table 3: Example for the solution

|  |  |  |  |
| --- | --- | --- | --- |
| **Process ID** | **Arrival Time** | **CPU Burst Time** | **Queue Number** |
| 0 | 0 | 4 | 0 |
| 1 | 1 | 5 | 1 |
| 2 | 2 | 3 | 0 |
| 3 | 3 | 6 | 2 |
| 4 | 4 | 2 | 1 |

Time 0: Executing Process 0 from Queue 0

Time 1: Executing Process 0 from Queue 0

Time 2: Executing Process 0 from Queue 0

Time 3: Executing Process 0 from Queue 0

Time 4: Executing Process 2 from Queue 0

Time 5: Executing Process 4 from Queue 1

Time 6: Executing Process 4 from Queue 1

Time 7: Executing Process 2 from Queue 0

Time 8: Executing Process 2 from Queue 0

Time 9: Executing Process 1 from Queue 1

Time 10: Executing Process 1 from Queue 1

Time 11: Executing Process 1 from Queue 1

Time 12: Executing Process 1 from Queue 1

Time 13: Executing Process 1 from Queue 1

Time 14: Executing Process 3 from Queue 2

Time 15: Executing Process 3 from Queue 2

Time 16: Executing Process 3 from Queue 2

Time 17: Executing Process 3 from Queue 2

Time 18: Executing Process 3 from Queue 2

Time 19: Executing Process 3 from Queue 2

\*\* Process exited - Return Code: 0 \*\*

Press Enter to exit terminal

The test results indicate the sequence of process executions over time based on the MLQ scheduler with SJF (Shortest Job First) and FCFS (First-Come-First-Served) principles. Let's analyze the output:

* Time 0 - 3: Process 0 from Queue 0 is the only available process at the beginning. It gets executed in a loop because its burst time is greater than the time quantum allocated for each execution.
* Time 4: Process 2 from Queue 0 is the next available process. It gets executed.
* Time 5 - 6: Process 4 from Queue 1 is the next available process. It gets executed, and the execution continues for the next time quantum.
* Time 7 - 8: Process 2 from Queue 0 gets another chance to execute.
* Time 9 - 13: Process 1 from Queue 1 is the next available process. It gets executed for several time quanta.
* Time 14 - 18: Process 3 from Queue 2 is the last available process. It gets executed for several time quanta.

The process execution is based on the combination of SJF and FCFS principles. In this output, the SJF queue (priority queue) prioritizes processes with the shortest remaining burst time. If multiple processes have the same remaining burst time, FCFS is used to break ties.

Overall, the scheduler ensures that processes are executed in an order that balances both the SJF and FCFS principles, considering the characteristics of each process and the available queues.

# 

# **10 Conclusion**

In conclusion, the implemented Multi-Level Queue (MLQ) scheduler with First-Come-First-Serve (FCFS) and Shortest Job First (SJF) logic within each priority queue, along with an aging mechanism, demonstrates the dynamics of process scheduling and priority adjustment over time. Here are key observations and conclusions based on the simulation:

**FCFS and SJF Logic**: The scheduling logic within each priority queue successfully follows FCFS and SJF principles. FCFS ensures that the first-arrived process in each queue is executed first, while SJF prioritizes processes based on their burst times within the same queue.

**Aging Mechanism:** The aging mechanism dynamically adjusts the priority of waiting processes, allowing lower-priority processes to gradually move to higher-priority queues over time. This feature helps prevent long-term starvation by providing opportunities for lower-priority processes to be executed.

**Execution Order:** The Gantt chart illustrates the execution order of processes and the time intervals during which each process is active. The scheduler effectively balances the execution of processes across different priority queues, demonstrating fairness in process execution.

**Starvation Mitigation:** While the aging mechanism contributes to fairness, it may not completely eliminate starvation, especially if there are significant differences in burst times among processes. Further refinements or more advanced scheduling algorithms may be necessary for comprehensive starvation mitigation.

**Flexibility and Extensibility:** The implemented architecture allows for flexibility and extensibility. Modifications and additions, such as adjusting priority adjustment strategies or incorporating additional scheduling algorithms, can be made to suit specific requirements or experiment with different scenarios.

**Real-World Considerations**: In a real-world scenario, the implemented solution is a simplified simulation. Real-world considerations, such as concurrent processing, error handling, and more sophisticated scheduling algorithms, would be crucial for handling the complexities of actual systems.

In summary, the MLQ scheduler provides a practical demonstration of process scheduling with FCFS and SJF logic within each priority queue, coupled with an aging mechanism. The simulation serves as a foundation for understanding basic scheduling principles and can be further enhanced for more advanced studies or real-world applications.

# **11 Future Enhancements**

The provided Multi-Level Queue (MLQ) scheduler simulation serves as a foundational example, and there are several ways to enhance and extend its functionality for further educational or practical purposes. Here are some suggestions for future enhancements:

* **Interactive User Interface**: Implement an interactive user interface to input process details, set parameters, and visualize the simulation in a more user-friendly manner.
* **Configurable Time Quantum:** Allow users to configure the time quantum for Round Robin scheduling in one or more priority queues, providing more flexibility in scheduling policies.
* **Dynamic Process Arrival:** Introduce dynamic process arrival during the simulation to emulate real-world scenarios where processes arrive over time.
* **Visualization Tools:** Implement graphical tools or charts to visualize the execution progress, Gantt charts, and other relevant metrics. Visualization aids in better understanding the scheduling dynamics.
* **Statistical Analysis:**Add statistical analysis features to collect and display performance metrics such as turnaround time, waiting time, and throughput. This would provide insights into the efficiency of the scheduling algorithm.
* **Advanced Scheduling Algorithms:** Extend the scheduler to include more advanced scheduling algorithms such as Priority Scheduling, Round Robin with Priority, or Multilevel Feedback Queue Scheduling. This allows for a more comprehensive comparison of different scheduling policies.
* **Concurrency and Parallelism:** Integrate concepts of concurrency and parallelism to simulate a more realistic environment where multiple processors are involved in process execution.
* **Error Handling and Robustness:** Implement robust error handling mechanisms to handle edge cases and unexpected scenarios. This ensures the stability and reliability of the simulation.
* **Dynamic Priority Adjustment:** Enhance the aging mechanism to dynamically adjust priorities based on factors like process behavior, historical data, or system load. This can contribute to more adaptive and intelligent scheduling.
* **Logging and Debugging:** Integrate logging mechanisms for better debugging and troubleshooting. Log critical events, process states, and any anomalies that occur during the simulation.
* **Real-Time Simulation:** Modify the simulation to run in real-time, allowing users to observe the scheduling decisions and process execution in a more dynamic and interactive way.
* **Integration with Real Systems:**Explore possibilities for integrating the simulation with real system data or interfaces. This could involve connecting the simulation to actual processes or obtaining input from external systems.

# **12 References**

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 5
2. <https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html>
3. <https://www.quora.com/>
4. Sarwar, Nadeem & Aslam, Naila & Batool, Amna. (2016). Designing a Model for improving CPU Scheduling by using Machine Learning. International Journal of Computer Science and Information Security,. 14. 201-204.
5. <https://www.prepbytes.com/blog/queues/multilevel-queue-mlq-cpu-scheduling/#:~:text=A%20multilevel%20queue%20scheduling%20algorithm,memory%20requirements%2C%20and%20CPU%20usage>.
6. <https://www.studytonight.com/operating-system/multilevel-feedback-queue-scheduling>
7. Muhsen, Babiceanu.; Systems Engineering Approach toCPU Scheduling for Mobile Multimedia Systems - 2011IEEE International Systems Conference, (SysCon), 2011
8. <https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/>
9. <https://medium.com/@barishasdemir/implementation-of-cpu-scheduling-algorithms-42786fef28f3>
10. <https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/>
11. http://www.ijmse.org/Volume5/Issue8/paper3.pdf

# **13 Appendix(Include the link to self reflection video of the project implementation)**