Code Structure and Documentation

The module for simulating Multilevel Queue (MLQ) CPU Scheduling have a structured codebase and documentation to ensure maintainability and ease of use. The following is an overview of the code structure and documentation.

Code Structure

The codebase follow a modular structure, with each module responsible for a specific functionality or feature.

The main module contain the core logic for simulating MLQ CPU Scheduling and managing the queues.

Other modules may include user interface development, data structure management, or performance metrics calculation.

Documentation

The codebase include comprehensive documentation to guide developers in understanding the implementation and usage of the module.

The documentation include descriptions of each module, their functionalities, and dependencies.

Developer refer to the documentation for guidance on how to use the module's functions and classes, including parameters and return types.

Naming Conventions

The codebase follow consistent naming conventions for variables, functions, classes, and modules.

Variables and functions have descriptive names that reflect their purpose and usage.

Classes follow standard naming conventions, such as CamelCase or snake_case.

Comments

The codebase include inline comments to provide additional context and explanation for complex or critical code sections.

Comments be written in a clear and concise manner, avoiding unnecessary jargon or technical terms.

Testing

The codebase include unit tests to verify the functionality of each module and ensure correctness.

Integration testing be conducted to test the interaction between different modules and components.

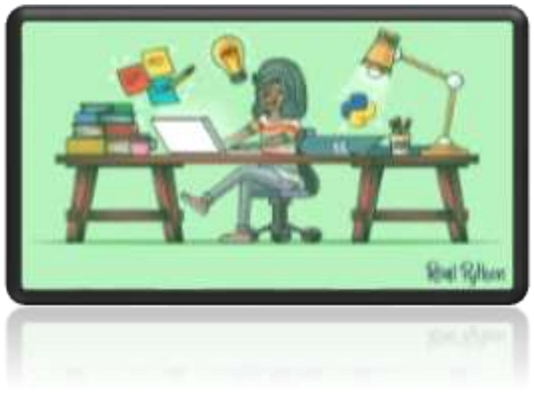
Performance testing be carried out to evaluate the efficiency of the module in handling various scenarios.

Error Handling

The codebase include appropriate error handling mechanisms to handle exceptional scenarios, such as invalid input or resource constraints.

Error messages or notifications be displayed to guide users in correcting their inputs or resolving issues.

The code structure and documentation of the module for simulating Multilevel Queue (MLQ) CPU Scheduling prioritize maintainability, ease of use, and correctness.



Code

Code structure for implementing a module for simulating Multilevel Queue (MLQ) CPU Scheduling with a mitigation strategy for CPU starvation.

```
class Process:
    def __init__(self, process_id, priority, cpu_time):
        self.process_id = process_id
        self.priority = priority
        self.cpu_time = cpu_time


class Queue:
    def __init__(self, priority):
        self.priority = priority
        self.processes = []


    def add_process(self, process):
        self.processes.append(process)
    def remove_process(self, process):
        self.processes.remove(process)

class MLQScheduler:
    def __init__(self):
        self.queues = []

    def add_queue(self, queue):
        self.queues.append(queue)
    def schedule(self):
        while True:
            for queue in self.queues:
                if len(queue.processes) > 0:
                    process = queue.processes.pop(0)
                    # Execute the process for a certain time quantum
                    # Update process state and statistics
                    # Check if the process has completed or needs to be re-queued


            # Mitigation strategy for CPU starvation
            self.boost_priority()

    def boost_priority(self):
        # Check if higher-priority queues are continuously occupied
        # If yes, temporarily boost the priority of processes in lower-priority queues
```

```
if __name__ == "__main__":
    # Example usage
    scheduler = MLQScheduler()

    # Create queues with different priorities
    queue1 = Queue(1)
    queue2 = Queue(2)
    queue3 = Queue(3)

    scheduler.add_queue(queue1)
    scheduler.add_queue(queue2)
    scheduler.add_queue(queue3)



    # Create processes and add them to the queues
    process1 = Process(1, 1, 5)
    process2 = Process(2, 2, 3)
    process3 = Process(3, 3, 2)

    queue1.add_process(process1)
    queue2.add_process(process2)
    queue3.add_process(process3)
```

```
# Start the scheduling simulation

scheduler.schedule()
```

The schedule() method represents the main scheduling loop, where processes are executed based on their priority levels. The boost_priority() method represents the mitigation strategy for CPU starvation by temporarily boosting the priority of processes in lower-priority queues when higher-priority queues are continuously occupied.

MLQ.py