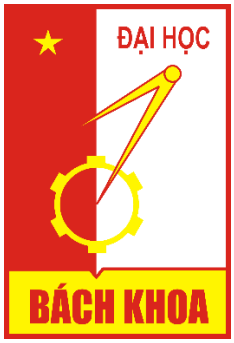
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

The School of Information and Communication Technology

----- ○ 🕮 ○ -----



Project Report
Course: Operating System

Topic: Multilevel Feedback Queue Scheduling

Supervisor: PhD.Đỗ Quốc Huy

Group members:

Trương Gia Bách	20210087
Trần Dương Chính	20210022
Phan Đức Hưng	20214903
Nguyễn Viết Minh	20214917

Hanoi, 1/2024

Table of Contents

I. Problem Introduction:

In the realm of operating systems, the efficiency of scheduling algorithms significantly impacts system performance. Multilevel feedback scheduling stands as a pivotal approach in managing process execution within a computer system. The intricacies of this algorithm lie in its dynamic nature, adapting priorities based on a process's behavior, execution time, and resource requirements.

The project aims to implement and demonstrate the functionality of a multilevel feedback scheduling algorithm. Through this implementation, the project seeks to shed light on the intricate workings of this approach, providing a practical illustration of its functioning within an operating system environment.

The primary objective is to design and develop a program that simulates the multilevel feedback scheduling algorithm. This program will showcase the dynamic nature of the algorithm, emphasizing its ability to adapt priorities, handle aging processes, and manage the execution of processes efficiently.

By implementing this program, the project aims to provide a comprehensive understanding of multilevel feedback scheduling, including its advantages, challenges, and real-world applicability in optimizing system performance and resource utilization.

Scope of the Project:

1. Algorithm Description:

Provide a comprehensive explanation of the multilevel feedback scheduling algorithm, outlining its core principles, priority adjustment mechanisms, and queue management strategies. This description will serve as the foundation for the implementation phase.

2. Algorithm Implementation:

Develop a program that embodies the multilevel feedback scheduling algorithm. Design the necessary data structures, queues, and scheduling logic to simulate the dynamic nature of the algorithm. Implement priority adjustments, aging processes, and context switching mechanisms.

3. Performance Evaluation:

Conduct rigorous testing and performance evaluation of the implemented algorithm. Evaluate its efficiency under varying workloads, considering factors like waiting time and average waiting time.

4. Graphical User Interface (GUI):

Create an intuitive and user-friendly GUI to interact with the scheduling algorithm simulation. The interface should allow users to input process details, visualize the scheduling decisions, and

display real-time updates of process execution. Incorporate graphical representations such as Gantt charts to aid in understanding the scheduling process.

II. Algorithm Description:

MLFQ endeavors to solve a dual challenge. Firstly, it aims to optimize turnaround time, a goal achievable through strategies like First Come First Serve or prioritizing shorter tasks in Shortest Job First. Unfortunately, the OS often lacks the exact duration knowledge required by algorithms like FCFS or SJF. Secondly, MLFQ strives to make systems responsive for interactive users, reducing response time. However, algorithms like Round Robin improve response time at the expense of turnaround time. Thus, the conundrum: lacking specific process information, how can we craft a scheduler to meet these objectives? How can it learn from running jobs and make smarter scheduling choices?

We focus on elucidating the fundamental algorithms underpinning a multi-level feedback queue—a pivotal step in constructing such a scheduler. While implemented MLFQs may differ in specifics, their approaches align. In this context, MLFQ encompasses distinct queues, each assigned varying priority levels. At any instance, a ready-to-run job resides in a single queue. The scheduler uses priorities to select the job for execution, favoring higher priority jobs from upper queues. In scenarios where multiple jobs share a queue with the same priority, round-robin scheduling comes into play. Hence, we establish the primary two rules for MLFQ:

Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t).

Rule 2: If Priority(A) = Priority(B), A & B run in RR.

The crux of MLFQ lies in its priority management approach. Rather than assigning fixed priorities, MLFQ dynamically adjusts a job’s priority based on observed behavior. For example, if a job frequently yields the CPU while awaiting keyboard input, MLFQ sustains its high priority, mimicking interactive process behavior. Conversely, if a job extensively uses the CPU, MLFQ reduces its priority. This adaptability allows MLFQ to learn about running processes, using their history to forecast future behavior.

MLFQ utilizes multiple queues arranged hierarchically, each with different priorities, ensuring time-critical tasks receive priority while also providing lower-priority processes with fair CPU time allocation.

i. Key Features:

Multiple queues: MLFQ maintains several queues with differing priorities.

Feedback mechanism: This mechanism adjusts process priorities based on past behavior. Processes completing their time slice in a lower-priority queue can be promoted to higher-priority queues for increased CPU time.

Time slicing: Each process is allocated a specific time quantum for execution within its current queue.

Dynamic priority adjustment: Process priorities change based on their behavior history, facilitating adaptability.

Preemption: High-priority processes can interrupt and take precedence over low-priority ones, ensuring they receive necessary CPU time.

ii. Functionality of MLFQ:

MLFQ scheduling involves setting parameters like the number of queues, time quantum, and priority adjustment. These parameters optimize resource usage and ensure fair process execution.

The sequence of steps in MLFQ scheduling upon a process's system entry:

Initially, a process is allocated to the highest priority queue.

The process operates within a specific time quantum in its current queue.

If the process finishes within the time quantum, it exits the system.

If not, it gets demoted to a lower priority queue with a shorter time quantum.

This promotion and demotion cycle continues, guided by process behavior.

High-priority queues take precedence, allowing lower-priority processes to run only when high-priority queues are empty.

The feedback mechanism facilitates process movement between queues based on their execution behavior.

This process persists until all processes are executed or terminated.

iii. Advantages and Disadvantages:

Advantages:

In modern computing systems, Multilevel Feedback Queue (MLFQ), a well-established instruction set algorithm for scheduling, offers several advantages:

Improved Response Time: MLFQ prioritizes quick execution by promptly addressing short tasks with the shortest waiting time, enhancing the reaction speed of processes.

Good Throughput: Using aging techniques, MLFQ ensures that longer-running processes receive sufficient CPU resources, allowing for balanced utilization and maximizing CPU usage.

Dynamic Priority Adjustment: MLFQ automatically adjusts process priorities based on their behavior. Less CPU-intensive tasks move to lower priority queues while CPU-intensive ones shift to higher priority queues.

Efficient CPU Utilization: MLFQ optimizes CPU usage by preempting processes when their allocated time quantum expires, preventing resource wastage.

Scalability: It efficiently manages a substantial volume of processes simultaneously, handling high workloads without compromising system performance.

Ease of Implementation: MLFQ's implementation is relatively straightforward, requiring specific data structures to manage multiple queues with varied priorities.

Disadvantages:

However, alongside its benefits, Multilevel Feedback Queue (MLFQ) presents some drawbacks:

Complexity: Managing multiple queues with distinct priorities and time quantum for each queue makes MLFQ a complex technique to implement and maintain compared to simpler scheduling approaches.

Priority Inversion: MLFQ might experience priority inversion, where a lower-priority process holds a resource required by a higher-priority process, causing the higher-priority process to wait.

Overhead: MLFQ incurs higher overhead due to the constant tracking and adjustment of process priorities within each queue, potentially leading to reduced performance.

Poor Predictability: The unpredictable nature of MLFQ scheduling makes accurately predicting process completion times challenging.

iv. Choosing Scheduler Parameters:

A multilevel feedback-queue scheduler's definition hinges on several key parameters:

Number of queues: Users can add queues to the scheduler in addition to a predefined First Come First Serve (FCFS) queue located at the scheduler's base.

Scheduling algorithm per queue: High priority queues follow a Round Robin-esque algorithm. Each queue is equipped with a quantum time denoting the CPU time each process can access at a stretch. Meanwhile, the bottom queue adheres to a First Come First Serve algorithm, catering to low-priority tasks like background processes.

Process upgrade to a higher priority queue: An approach involves monitoring the time a process spends waiting in the low-priority FCFS queue. After a predetermined threshold, these processes are promoted to the top priority queue.

Process demotion to a lower priority queue: Within high-priority queues, a Round Robin-like algorithm assists in downgrading processes when they exhaust their quantum time. This mechanism ensures fair distribution of running time among all processes.

Queue assignment for a process seeking service: Initially, high-priority processes are positioned in top queues, and lower-priority ones in bottom queues. However, in our setup, all arriving processes are initially placed in the top queue, leveraging the existing prioritization mechanism.

III. Algorithm Implementation:

i. Input:

Queue Input: Input is exclusively required for Round Robin queues, involving setting the queue ID and its respective quantum time.

```
class Queue:
    def __init__(self, queue_id, quantum):
        self.queue_id = queue_id
        self.quantum = quantum
        self.queue = []
```

Process Input:

For each process added to the scheduler, setting a process ID and acquiring information about its arrival and burst time is necessary.

```
class Process():
    def __init__(self, process_id, arrival_time, burst_time):
        self.process_id = process_id
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.remain_time = burst_time
        self.completion_time = -1
        self.waiting_time = 0
        self.remain_cpu = 0
        self.fcfs_arrive = 0
        self.cur_queue = 0

    # def __eq__(self, other):
    #     return self.arrival_time == other.arrival_time and self.process_id == other.process_id

    def __lt__(self, other):
        return (self.arrival_time, self.process_id) < (other.arrival_time, other.process_id)

    def get_name(self):
        return f"P{self.process_id}"
```

ii. Output:

From the provided queue structure and added processes, the desired output comprises:

A table displaying the state of all processes at each time step, showcasing:

Process name

Current queue of the process

Arrival time of the process

Burst time of the process

Remaining execution time of the process

Completion time of the process

Waiting time of the process

The Gantt Chart illustrating the scheduler's execution while processing the given tasks.

Calculation of the Average Waiting Time.

iii. Algorithm:

At each time step, one of the following events occurs:

Arrival of a process at the top queue.

```
# Add a new arriving process
def add_new_process(process, curtime):
    global last_timestep
    for proc in allprocess_list:
        # Update process with CPU
        if proc is process_with_cpu:
            proc.remain_time -= curtime-last_timestep
            self.table.insert(row=proc.process_id+1, column=4, value=proc.remain_time)

            proc.remain_cpu -= curtime-last_timestep

        # Increase other processes waiting time
        elif proc.arrival_time < curtime and proc.completion_time == -1:
            proc.waiting_time += curtime-last_timestep
            self.table.insert(row=proc.process_id+1, column=6, value=proc.waiting_time)

    # Add arriving process to 1st queue
    queue_list[0].queue.append(process)
    process.cur_queue = 0
    self.table.insert(row=process.process_id+1, column=1, value="RR1")
```

A process completes its CPU usage time, either downgrading to the lower queue or finishing its execution. A new process is selected to receive the CPU.

```
# Downgrade a process after using CPU
def downgrade(process, curtime):
    global last_timestep, process_with_cpu
    for proc in allprocess_list:
        # Update downgrade process
        if proc is process:
            queue_list[proc.cur_queue].queue.pop(0)
            proc.cur_queue += 1

            # Remain Time
            proc.remain_time -= proc.remain_cpu
            self.table.insert(row=proc.process_id+1, column=4, value=proc.remain_time)

            # Finish process
            if proc.remain_time == 0:
                self.table.insert(row=proc.process_id+1, column=1, value="X")
                proc.completion_time = curtime
                self.table.insert(row=proc.process_id+1, column=5, value=proc.completion_time)
                process_with_cpu = None
            else:
                # Update State
                if proc.cur_queue == len(queue_list)-1:
                    self.table.insert(row=proc.process_id+1, column=1, value="FCFS")
                    proc.fcfs_arrive = curtime
                else:
                    self.table.insert(row=proc.process_id+1, column=1, value=f"RR{proc.cur_queue+1}")

                # Move to below queue
                queue_list[proc.cur_queue].queue.append(proc)

        # Increase other processes waiting time
        elif proc.arrival_time < curtime and proc.completion_time == -1:
            proc.waiting_time += curtime-last_timestep
            self.table.insert(row=proc.process_id+1, column=6, value=proc.waiting_time)
```

```
# Give the CPU to a new process
def give_cpu(curtime):
    global last_cpu, process_with_cpu
    for qqueue in queue_list:
        if qqueue.queue:
            process = qqueue.queue[0]

            process.remain_cpu = min(process.remain_time, qqueue.quantum)
            self.table.insert(row=process.process_id+1, column=1, value="CPU")
            process_with_cpu = process

            # Update Gantt Chart
            if process is not last_cpu:
                if last_cpu != None:
                    process_text.set(process_text.get()+f"  P{last_cpu.process_id}  |")
                    time_text = f"{curtime}"
                    while len(time_text) < 7:
                        time_text = " "+time_text
                    timeline_text.set(timeline_text.get()+time_text)
                last_cpu = process
            return
```

A process waits excessively in the bottom FCFS queue and is promoted to the top queue.

```
# Upgrade a process from FCFS queue
def upgrade(process, curtime):
    global last_timestep
    for proc in allprocess_list:
        # Update process with CPU
        if proc is process_with_cpu:
            proc.remain_time -= curtime-last_timestep
            self.table.insert(row=proc.process_id+1, column=4, value=proc.remain_time)

            proc.remain_cpu -= curtime-last_timestep

        # Increase other processes waiting time
        elif proc.arrival_time < curtime and proc.completion_time == -1:
            proc.waiting_time += curtime-last_timestep
            self.table.insert(row=proc.process_id+1, column=6, value=proc.waiting_time)

    # Move process to high priority queue
    queue_list[0].queue.append(process)
    process.cur_queue = 0
    self.table.insert(row=process.process_id+1, column=1, value="RR1")
```

Considering the preemptive nature of the scheduler, if a higher priority process arrives while a lower priority process is executing, the CPU switches to the arriving process.

Each time step involves assessing these scenarios to update the tracking table with new information.

IV. Graphic User Interface:

i. Input:

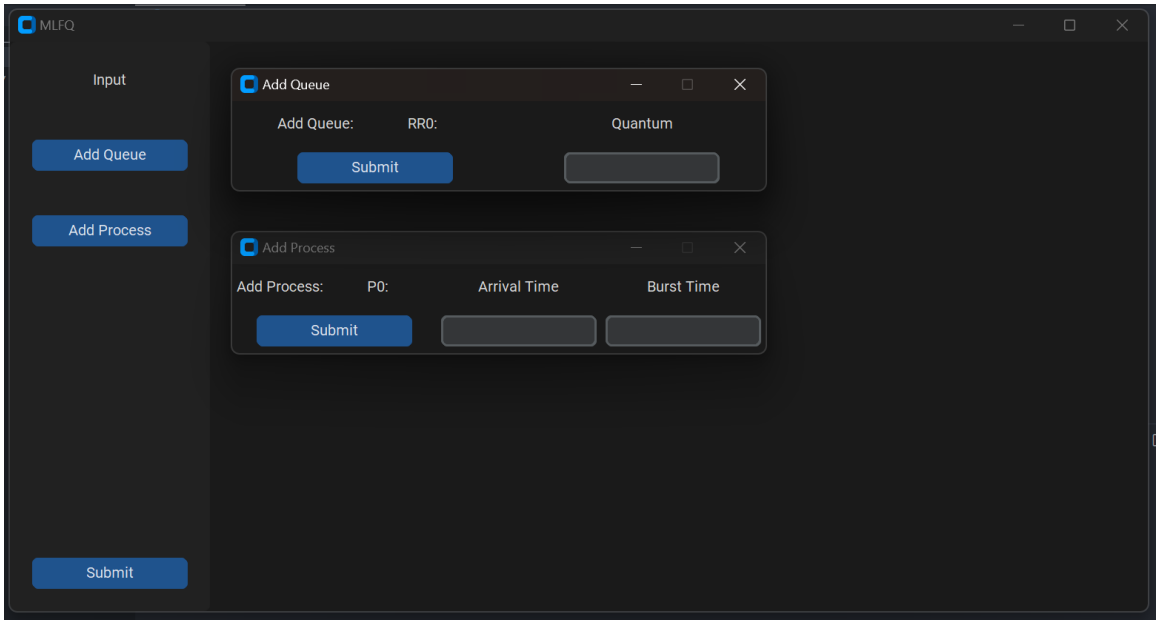
User can click the button “Add Queue” to add a new queue. Next, we need to type in the quantum for the queue then press “Submit”. This can be done with an “Add Queue” window.

```
# Add queue window
def add_queue_window():
    addqueueWindow = ctk.CTk()

    addqueueWindow.geometry("480x80+500+180")
    addqueueWindow.resizable(0, 0)
    addqueueWindow.title("Add Queue")

    addqueueWindow.grid_columnconfigure((0, 1), weight=1)
    addqueueWindow.grid_rowconfigure((0, 1), weight=1)

    # Label
    addqueue_label = ctk.CTkLabel(addqueueWindow, text=f"Add Queue:\tRR{queue_cnt}:\t")
    addqueue_label.grid(row=0, column=0)

    addqueue_quantum = ctk.CTkLabel(addqueueWindow, text="Quantum")
    addqueue_quantum.grid(row=0, column=1)

    # Entry
    addqueue_entry = ctk.CTkEntry(addqueueWindow)
    addqueue_entry.grid(row=1, column=1)

    # Add queue
    def add_queue():
        global queue_cnt
        queue_list.append(Queue(queue_cnt, int(addqueue_entry.get())))
        queue_cnt += 1
        addqueueWindow.destroy()

    # Submit button
    addqueue_button = ctk.CTkButton(addqueueWindow, text="Submit", command=add_queue)
    addqueue_button.grid(row=1, column=0)

    addqueueWindow.mainloop()
```

Similarly, user can use the “Add Process” window to add a new process. This time, the arrival time and the burst time of the process are required.

```python
# Add process window
def add_process_window():
    addprocessWindow = ctk.CTk()

    addprocessWindow.geometry("480x80+500+400")
    addprocessWindow.resizable(0, 0)
    addprocessWindow.title("Add Process")

    addprocessWindow.grid_columnconfigure((0, 1, 2), weight=1)
    addprocessWindow.grid_rowconfigure((0, 1), weight=1)

    # Label
    addprocess_label = ctk.CTkLabel(addprocessWindow, text=f"Add Process:\tP{process_cnt}:\t")
    addprocess_label.grid(row=0, column=0)

    addprocess_arrivaltime = ctk.CTkLabel(addprocessWindow, text="Arrival Time")
    addprocess_arrivaltime.grid(row=0, column=1)

    addprocess_bursttime = ctk.CTkLabel(addprocessWindow, text="Burst Time")
    addprocess_bursttime.grid(row=0, column=2)

    # Entry
    addprocess_entry_arrivaltime = ctk.CTkEntry(addprocessWindow)
    addprocess_entry_arrivaltime.grid(row=1, column=1)

    addprocess_entry_bursttime = ctk.CTkEntry(addprocessWindow)
    addprocess_entry_bursttime.grid(row=1, column=2)

    # Add process
    def add_process():
        global process_cnt
        process_list.append(Process(process_cnt, int(addprocess_entry_arrivaltime.get()), int(addprocess_entry_bursttime.get())))
        process_cnt += 1
        addprocessWindow.destroy()

    # Submit button
    addprocess_button = ctk.CTkButton(addprocessWindow, text="Submit", command=add_process)
    addprocess_button.grid(row=1, column=0)

    addprocessWindow.mainloop()
```

The input interface is given below:



ii. Output:

After the “Submit” button is pressed, the main frame of the GUI is displayed with a tracking time label, a “Next” button, a table to show processes’ information, a Gantt Chart. The time track, the table and the Gantt Chart is updated every time the user press “Next” to display the output at consecutive time stamp.

In the end the Average Waiting Time is calculated and the final output can be seen below:

V. Contribution:

Trương Gia Bách 20210087	GUI, report, slide
Trần Dương Chính 20210122	Downgrade a process, report
Phan Đức Hưng 20214903	Arrival of new process
Nguyễn Viết Minh 20214917	Upgrade a process, report

VI. References:

[1] Customtkinter. (n.d.). PyPI. https://pypi.org/project/customtkinter/0.3/

[2] CTkTable. (n.d.). PyPI. https://pypi.org/project/CTkTable/