# Dynamic Scheduling Without Real Time Requirement

MOHAMMAD ASHRAFUZZAMAN SIDDIQI
*Electronic Engineering*
*University of Applied Sciences Hamm Lippstadt*
Dortmund, Germany
mohammad-ashrafuzzaman.siddiqi@stud.hshl.de

***Abstract*—Scheduling is a fundamental function in computer and electronic systems that manages the optimal allocation of resources such as CPU time, memory, and I/O devices to different activities. This study investigates dynamic scheduling, which allocates resources in real time based on the system's current state, as opposed to static scheduling, which pre-allocates resources at compile time. This paper focuses on dynamic scheduling without the constraints of real-time needs, emphasising its importance in increasing throughput, optimising resource utilisation, and improving system responsiveness in non-real-time settings. Key dynamic scheduling algorithms such as Round Robin, Shortest Remaining Time First (SRTF), and Multilevel Feedback Queue (MLFQ) are examined, with a focus on their implementation and applications in cloud computing, manufacturing, energy harvesting systems, IoT, and grid computing. By utilising these techniques, systems can attain flexibility and efficiency while responding to changing workloads and circumstances. The report finishes by discussing future research topics and potential breakthroughs in dynamic scheduling approaches.**

***Index Terms*—Scheduling, static, dynamic, real time.**

## I. Introduction

Scheduling is a fundamental process in computing and electronic systems, involving the allocation of resources to tasks over time to achieve specific objectives. The resources that can be assigned include CPU time, memory, and I/O devices. Scheduling in computing technology serves the primary goal of optimising metrics such as throughput and latency and ensuring efficient and fair usage of all available resources to complete all tasks [1].

This study will investigate the many types of scheduling, providing examples and highlighting different techniques used in dynamic scheduling. The emphasis will be on how dynamic scheduling operates without the strict timing constraints of a real-time system, allowing for a consideration of what applications dynamic scheduling can be beneficial for in this context. Future ideas and views will be revealed in the last section before drawing conclusions on the many issues investigated during my research.

Generally there are two types of scheduling approach namely Dynamic and Static scheduling.

Identify applicable funding agency here. If none, delete this.

1) **Static Scheduling:** This is a process that involves the predetermined allocation of resources to tasks, and this is done before the system is executed. Static scheduling involves creating a schedule plan at compile time. For this to be possible, prior knowledge of task characteristics such as execution times, dependencies, and communication requirements must all be readily available [2]. Static scheduling trades flexibility (the ability to make changes at runtime) for simplicity. The simplicity of the system also allows it to be optimised more effectively. Static scheduling is often used in environments where tasks and resource availability are predictable and well-defined.Static scheduling is used in areas such as:
   - Embedded systems : where task are scheduled based on fixed time slots.
   - Batch Processing Systems: Where jobs are scheduled to run in a predefined order based on prior knowledge of resource requirements.

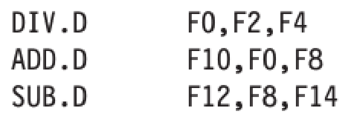
   While static scheduling is simple and can be optimised for known conditions, it is inflexible and unsuitable for systems in which tasks or resource availability change dynamically.

## II. Dynamic Scheduling

Dynamic scheduling refers to a process in which tasks and resources are assigned in real-time, either after compilation or during the execution of the system. These decisions are continuous and allow for adaptation and flexibility to varied systemic changes. The state of the system is consistently monitored, and any adjustment to resource allocation is made when required to optimise the processor's efficacy and performance. Dynamic scheduling is a hardware-based approach, and it is used mostly when the dependencies are not known at compile time. Dynamic scheduling is essential in environments characterized by variability and unpredictability. It allows systems to respond to changes such as new task arrivals, resource failures, or shifting priorities. This flexibility is crucial to maintaining efficiency and ensuring that resources are utilised effectively [2].

### *A. Scoreboarding*

This is a technique that allows the instruction to execute out of order where structural hazards and data dependencies do not exist. Scoreboarding can only function when sufficient resources exist. It allows instruction-level parallelism within a CPU pipeline [3].



```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F12,F8,F14
```

Fig. 1. Instruction Set [3]

The scoreboard ensures correct execution by monitoring resource and data dependencies. The DIV.D instruction executes first, as there are no dependencies. ADD.D waits for DIV.D to complete and write its result to F0, avoiding read-after-write (RAW) hazards. Finally, SUB.D waits for ADD.D to complete and write its result to F10. This sequence ensures that each instruction is executed in the correct order without causing hazards [3].

**Execution Timeline**:

- **Cycle 1**: Issue DIV.D.
- **Cycle 2**: DIV.D executes.
- **Cycle 3**: DIV.D writes to F0.
- **Cycle 4**: Issue ADD.D.
- **Cycle 5**: ADD.D executes.
- **Cycle 6**: ADD.D writes to F10.
- **Cycle 7**: Issue SUB.D.
- **Cycle 8**: SUB.D executes.
- **Cycle 9**: SUB.D writes to F12.

### *B. Tomasulo Algorithm*

This is a hardware-dependent resolution scheme for implementing dynamic scheduling and thereby enhancing parallelism. It was invented by Robert Tomasulo [3]. The algorithm uses register renaming to remove the name dependency, reducing WAR (Write After Read) and WAW (Write After Write) while tracking available operands to satisfy the data dependency. It also employs a Common Data Bus (CDB) to broadcast results to all reservation stations to update operands as soon as they are available [3].

Using the same instruction set as 1, tomasulo execution is as follows:

**Execution Steps:**

- **DIV.D F0, F2, F4**:
  - **Issue**: The instruction is issued to a reservation station if available. Operands F2 and F4 are read if ready.
  - **Execute**: The division operation is performed when both operands are available.
  - **Write-Back**: The result is written back to F0 via the CDB, updating any waiting instructions.
- **ADD.D F10, F0, F8**:
  - **Issue**: The instruction is issued to a reservation station. F0 is read directly, but F8 waits for the result of DIV.D to become available.
  - **Execute**: The addition operation is performed once F0 is available, which occurs after the result of DIV.D is written to F0.
  - **Write-Back**: The result of the addition operation is written back to register F10 via the CDB, updating any waiting instructions.
- **SUB.D F12, F10, F14**:
  - **Issue**: The instruction is issued to a reservation station. F10 is read directly, but F14 waits for the result of ADD.D.
  - **Execute**: The subtraction operation is performed once F10 is ready.
  - **Write-Back**: The result is written back to F12 via the CDB.

**Execution Timeline**:

- **Cycle 1**: Issue DIV.D to a reservation station.
- **Cycle 2**: DIV.D executes.
- **Cycle 3**: DIV.D writes to F0.
- **Cycle 4**: Issue ADD.D to a reservation station.
- **Cycle 5**: ADD.D executes.
- **Cycle 6**: ADD.D writes to F10.
- **Cycle 7**: Issue SUB.D to a reservation station.
- **Cycle 8**: SUB.D executes.
- **Cycle 9**: SUB.D writes to F12.

## III. Dynamic Scheduling without real time requirement

Real-time systems, or real time, denote a sense of urgency or time limit in computing; it infers that there is a time constraint for which an input or external event gets a response; this time constraint is often termed a deadline. Dynamic scheduling plays a crucial role in real-time systems; it helps in the effective allocation of resources and tasks to meet a deadline. In these systems, it also helps to address the unpredictability of a system by adapting to varying workloads and other different conditions while still figuring out the most efficient way to process instruction. However, it is interesting to ponder how dynamic scheduling can function without the confines of a deadline or more general real-time requirements; this is the main focus of our paper.

Dynamic scheduling without a real-time requirement can play an important role in enhancing the overall performance of a computing environment. Its primary goal, when not concerned with deadlines or real-time constraints, becomes functions such as maximising throughput, optimising resource utilisation, and being responsive.

- **Maximizing Throughput:** Throughput can be defined as the number of tasks that are able to be completed in a given time period. Dynamic scheduling can drastically help increase the throughput in a system by balancing the workload distribution amongst multiple processors while also adjusting task priorities based on the current system conditions and status. Load Balancing as discussed by [3] is a technique that accomplishes this, it is a dynamic scheduling algorithm that distributes tasks evenly, preventing bottlenecks and as a result increasing overall task completion rate in parallel computing systems.

- **Resource Utilization:** Every system includes resources such as CPU, memory and Input/ Output devices. Dynamic scheduling algorithms can help in the optimization of resource utilization to enhance overall system performance and reduce idle times. The dynamic scheduling algorithms allocate resources to different tasks based on their current requirements and the availability of resources. This helps in minimizing idle times and avoiding resource contention. The nature of these dynamic algorithms also ensures adaptation to changes in a number of available resources such as additional memory storage and ensures the system performs efficiently
- **Improving Responsiveness** In non-real-time systems, the user experience and system responsiveness are critical. Dynamic scheduling enhances responsiveness by prioritising tasks requiring quick responses, such as user inputs or interactive procedures. This entails prioritising lower-priority tasks in favour of those that require immediate attention. In [5] it is demonstrated how dynamic priority adjustment strategies can increase system responsiveness by guaranteeing that high-priority tasks are completed quickly while preserving overall system efficiency [5].

## Algorithms in Dynamic Scheduling without Real-Time Requirements

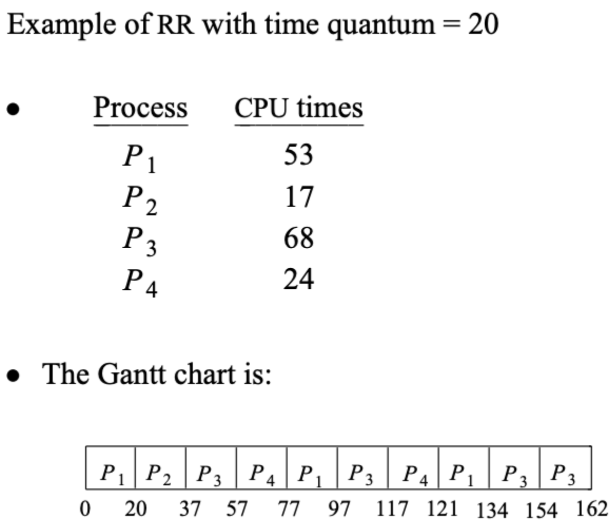
Dynamic scheduling algorithms stand out due to their adaptability. Unlike static scheduling, where decisions are made prior to execution based on a fixed plan, dynamic scheduling algorithms make real-time decisions based on the current state of the system. This allows them to adjust to changing workloads and system conditions, offering greater flexibility and improved performance.

This paper describes several key dynamic scheduling algorithms, highlighting their principles, applications, and implementation::

1) **Round Robin (RR):** Round Robin scheduling is a preemptive approach (processes can be terminated in the event of a higher priority) in which each process in the system is assigned a fixed amount of quantum time, often between 10-100 milliseconds [5]. After this time period, the process is pre-empted and added to the end of the queue; the scheduler also cycles through the processes and allocates CPU time to each for the stated quantum duration [5].
In this figure 2 the round-robin scheduling algorithm with a quantum time of 20 units is illustrated. 4 Processes P1,P2, P3 and P4 have varying assigned cpu burst times [5].

### *A. Execution Process*

**Initial Allocation:**


Example of RR with time quantum = 20

| Process | CPU times |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

- The Gantt chart is:

Fig. 2. [5]

- $P_1$ executes 20 units (33 remaining).
- $P_2$ executes 17 units (0 remaining).
- $P_3$ executes 20 units (48 remaining).
- $P_4$ executes 20 units (4 remaining).

**Second Round:**

- $P_1$ resumes 20 units (13 remaining).
- $P_3$ resumes 20 units (28 remaining).
- $P_4$ completes remaining 4 units (0 remaining).

**Third Round:**

- $P_1$ completes remaining 13 units (0 remaining).
- $P_3$ resumes 20 units (8 remaining).

**Final Round:**

- $P_3$ completes remaining 8 units (0 remaining).

The Gantt chart visually represents the execution order and time allocation of each process: The timeline starts at 0 and increments by the time quantum or the remaining burst time, whichever is smaller. Each block in the Gantt chart shows which process is running at a given time interval.

### *B. Implementation*

The provided code implements the Round Robin CPU scheduling algorithm to manage the execution of processes on a CPU.

```
from collections import deque

def round_robin(processes, quantum):
    queue = deque(processes)
    time = 0
```

The ‘round_robin‘ function initializes a queue using the deque data structure to hold processes and initializes the time variable.

```
while queue:
    process = queue.popleft()
```

The algorithm enters a loop that continues until the queue is empty. In each iteration, a process is dequeued from the front of the queue.

```
if process['burst_time'] > quantum:
    time += quantum
    process['burst_time'] -= quantum
    queue.append(process)
else:
    time += process['burst_time']
    process['burst_time'] = 0
```

If the burst time of the process is greater than the quantum time, it is executed for the quantum time, and its burst time is reduced accordingly. Otherwise, it is executed for its remaining burst time.
During each iteration, the current time and the ID of the processed process are printed to track the progress.

```
processes = [{'id': 1, 'burst_time': 10
             {'id': 2, 'burst_time': 5}
             {'id': 3, 'burst_time': 8}
round_robin(processes, quantum=4)
```

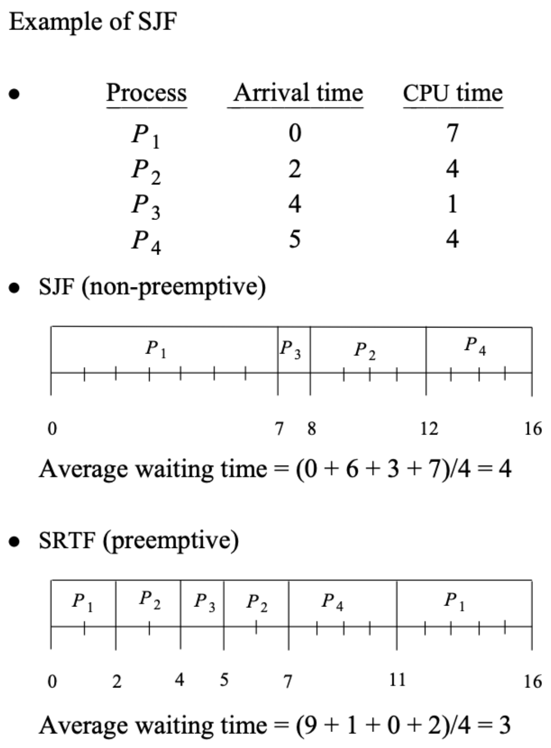
2) **Shortest Remaining Time First (SRTF):** This is a preemptive version of the shortest job next algorithm (SJN). It is important to note that the SJN algorithm serves the purpose of reducing bias, which favours longer processes in an algorithm like the FCFS. For the shortest time remaining first algorithm, the scheduler always selects the process or task with the shortest remaining processing time or the shortest amount of time to complete.
The following image in 3 illustrates an example of both SJN and SRTF scheduling algorithms:
**Explanation:**
In this example, we have four processes with their respective arrival and CPU times:
   - Process $P_1$: Arrival time = 0, CPU time = 7
   - Process $P_2$: Arrival time = 2, CPU time = 4
   - Process $P_3$: Arrival time = 4, CPU time = 1
   - Process $P_4$: Arrival time = 5, CPU time = 4

   **SJF (non-preemptive):**
   In the non-preemptive SJF scheduling, the processes are executed in the following order:
   - $P_1$ executes first from time 0 to 7.
   - $P_3$ executes next from time 7 to 8 (since it arrives at time 4 and has the shortest CPU time of 1).
   - $P_2$ executes from time 8 to 12.
   - Finally, $P_4$ executes from time 12 to 16.

   The average waiting time is calculated as:

$$\text{Average waiting time} = \frac{(0+6+3+7)}{4} = 4$$

   **SRTF (preemptive):**


Example of SJF

| Process | Arrival time | CPU time |
|---|---|---|
| $P_1$ | 0 | 7 |
| $P_2$ | 2 | 4 |
| $P_3$ | 4 | 1 |
| $P_4$ | 5 | 4 |

- SJF (non-preemptive)

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

- SRTF (preemptive)

Average waiting time = (9 + 1 + 0 + 2)/4 = 3

Fig. 3. SJF and SRTF Scheduling [5]

In the preemptive SRTF scheduling, the processes are executed in the following order:
- $P_1$ starts execution at time 0.
- $P_2$ arrives at time 2 and preempts $P_1$ because its CPU time (4) is less than the remaining time of $P_1$ (5).
- $P_3$ arrives at time 4 and preempts $P_2$ because its CPU time (1) is the shortest.
- After $P_3$ completes, $P_2$ resumes execution.
- $P_4$ arrives at time 5 but does not preempt $P_2$ (remaining time = 2).
- Once $P_2$ finishes, $P_4$ executes next.
- Finally, $P_1$ completes its remaining time.

The average waiting time is calculated as:

$$\text{Average waiting time} = \frac{(9+1+0+2)}{4} = 3$$

This example demonstrates that the SRTF algorithm further reduces the average waiting time compared to the non-preemptive SJF algorithm by dynamically selecting the process with the shortest remaining time. This goes to further highlight the importance of dynamic scheduling algorithms.

## C. Implementation

The Shortest Remaining Time First (SRTF) algorithm is a preemptive scheduling algorithm that selects the process with the shortest remaining CPU time for execution.

The following steps outline the SRTF implementation in Python:

a) Process Definition: Each process is represented by a class containing attributes such as id, arrival_time, burst_time, remaining_time, and completion_time A.
b) Initialization: Initialize the system clock (time), an empty list for completed processes, and the ready queue A.
c) Process Arrival and Sorting: At each time unit, newly arrived processes are added to the ready queue, which is then sorted by remaining time to ensure the process with the shortest remaining time is at the front A.
d) Process Selection and Execution: The scheduler selects the process with the shortest remaining time from the ready queue. If the current process has more remaining time than the process at the front of the queue, it is preempted and placed back in the queue A.
e) Process Completion: The selected process executes for one time unit. If it completes, it is marked as finished and added to the list of completed processes A.
f) Waiting Time Calculation: After all processes have completed, the waiting time for each process is calculated as the difference between its completion time and the sum of its arrival and burst times. The average waiting time is then computed A.

This concise explanation summarizes the key steps involved in implementing the SRTF scheduling algorithm in Python. The full implementation is provided in the appendix for reference A.

3) **Multilevel Feedback Queue (MLFQ):** This is a sophisticated scheduling algorithm used in different operating systems to optimise process scheduling and improve general system performance [**?**]. This approach addresses the limitations of simpler scheduling algorithms such as FCFS, SJF, RR, and Priority Scheduling by introducing dynamic prioritization and feedback mechanisms. For example, in FCFS (First Come, First Serve), a process that requires a short CPU time can be placed at the back of the queue, which in turn reduces the efficiency of the system. The multilevel feedback queue addresses limitations like this and others by implementing multiple queues, each with a different priority level and its own scheduling algorithm [5]. It places different processes in multiple queues based on their behaviour and execution history. Processes can move between queues based on recent CPU burst times [5].

- **Multiple Queues**: MLFQ maintains several queues, each with its own priority level. Processes in higher-priority queues are given preference over those in lower-priority queues.
- **Dynamic Priority Adjustment**: Unlike static priority algorithms, MLFQ dynamically adjusts the priority of processes based on their behavior and CPU burst characteristics. This helps balance the needs of both CPU-bound and I/O-bound processes.
- **Aging**: To prevent starvation of lower-priority processes, MLFQ employs an aging mechanism. If a process waits too long in a lower-priority queue, its priority is gradually increased, ensuring it eventually gets CPU time.
- **Feedback Mechanism**: Processes that use a lot of CPU time are moved to lower-priority queues, while processes that frequently yield the CPU (e.g., I/O-bound processes) can be promoted to higher-priority queues. This feedback mechanism helps achieve a balance between responsiveness and throughput.
- **Time Quanta**: Each queue can have a different time quantum, the amount of CPU time allocated to each process before it is preempted. Typically, higher-priority queues have shorter time quanta, and lower-priority queues have longer time quanta.
- **Flexibility**: MLFQ is highly flexible and can be tuned to meet specific system requirements by adjusting the number of queues, their priority levels, and the time quanta.

## *D. Example of MLFQ*

Consider a system with three queues:

- **Queue 0 (Q0)**: Highest priority, time quantum of 8 milliseconds.
- **Queue 1 (Q1)**: Medium priority, time quantum of 16 milliseconds.
- **Queue 2 (Q2)**: Lowest priority, using FCFS (no time quantum).

The scheduling process for these queues follows:

a) A new job enters Queue 0.
b) If it does not finish within 8 milliseconds, it is moved to Queue 1.
c) If it does not finish within 16 milliseconds in Queue 1, it is moved to Queue 2.
d) Queue 2 processes jobs using FCFS without preemption.

Let's consider three processes:

- **Process** $P_1$: Burst time = 20 milliseconds.
- **Process** $P_2$: Burst time = 18 milliseconds.
- **Process** $P_3$: Burst time = 15 milliseconds.

## *E. Python Implementation Explanation*

The full code can be found in the appendix B

a) **Process Class**:
   - The `Process` class encapsulates the process attributes: `id`, `burst_time`, `remaining_time`, and `queue_level`.
b) **MLFQ Scheduling Function**:

- `mlfq_scheduling` initializes three queues and assigns each process to the highest priority queue initially.
- The function iterates through the queues, processing each one according to its time quantum.
- If a process does not complete within its allotted time quantum, it is demoted to the next lower-priority queue.
- The function prints the current time, process id, queue level, and remaining time for each process.

c) **Execution Logic**:

- Processes in higher-priority queues (Q0 and Q1) are preempted if they exceed their time quantum and moved to a lower-priority queue.
- Processes in the lowest-priority queue (Q2) are processed using FCFS without preemption.

The full implementation is provided in the appendix for reference B.

## References

[1] M. L. Pinedo, "Scheduling," Springer, vol. 29, 2012.

[2] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," ACM Computing Surveys (CSUR), vol. 31, no. 4, pp. 406–471, 1999.

[3] J. L. Hennessy and D. A. Patterson, "Computer architecture: a quantitative approach," Elsevier, 2011.

[4] A. Tanenbaum, "Modern operating systems," Pearson Education, Inc., 2009.

[5] A. Silberschatz, J. L. Peterson, and P. B. Galvin, "Operating system concepts," Addison-Wesley Longman Publishing Co., Inc., 1991.

[6] W. Stallings, "Operating Systems: Internals and Design Principles," Prentice Hall Press, 2011.

## Appendix

### A. Full Code for SRTF Implementation

Listing 1. Python code for SRTF scheduling

```
class Process:
    def __init__(self, id, arrival_time, burst_time):
        self.id = id
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.remaining_time = burst_time
        self.completion_time = 0

def srtf(processes):
    time = 0
    completed_processes = []
    ready_queue = []
    current_process = None
    last_process = None

    while len(completed_processes) < len(processes):
        for process in processes:
            if process.arrival_time == time:
                ready_queue.append(process)
        ready_queue.sort(key=lambda x: x.remaining_time)

        if current_process and ready_queue:
            if current_process.remaining_time > ready_queue[0].remaining_time:
                ready_queue.append(current_process)
                ready_queue.sort(key=lambda x: x.remaining_time)
                current_process = ready_queue.pop(0)

        if not current_process and ready_queue:
            current_process = ready_queue.pop(0)

        if current_process:
            current_process.remaining_time -= 1
            if current_process.remaining_time == 0:
                current_process.completion_time = time + 1
                completed_processes.append(current_process)
                current_process = None
        else:
            time += 1

    total_waiting_time = 0
    for process in completed_processes:
        waiting_time = process.completion_time - process.arrival_time - process.burst_time
        total_waiting_time += waiting_time

    average_waiting_time = total_waiting_time / len(completed_processes)
    print(f"Average■Waiting■Time:■{average_waiting_time}")

processes = [
    Process(1, 0, 7),
    Process(2, 2, 4),
    Process(3, 4, 1),
    Process(4, 5, 4)
]

srtf(processes)
```

### *B. Full Code for MLFQ Implementation*

Listing 2. Python code for MLFQ scheduling

```
from collections import deque

class Process:
    def __init__(self, id, burst_time):
        self.id = id
        self.burst_time = burst_time
        self.remaining_time = burst_time
        self.queue_level = 0

def mlfq_scheduling(processes, time_quantum):
    queues = [deque() for _ in range(3)]
    for process in processes:
        queues[0].append(process)

    time = 0
    while any(queues):
        for i in range(len(queues)):
            if queues[i]:
                process = queues[i].popleft()
                quantum = time_quantum[i]
                execution_time = min(process.remaining_time, quantum)
                process.remaining_time -= execution_time
                time += execution_time

                print(f"Time:■{time},■Process:■{process.id},■Queue■Level:■{i},■Remaining■Time:

                if process.remaining_time > 0:
                    if i < len(queues) - 1:
                        process.queue_level += 1
                    queues[process.queue_level].append(process)
                break
        else:
            time += 1

processes = [
    Process(1, 20),
    Process(2, 18),
    Process(3, 15)
]

time_quantum = [8, 16, float('inf')]
mlfq_scheduling(processes, time_quantum)
```