Operating System

Lecturer: Dr. Do Quoc Huy

Class ID: 151995

Scheduling Algorithms in Multiprocessor Systems

AUTHORS

Vu Hai Dang - 20225962
Nguyen Minh Khoi - 20226050
Le Dai Lam - 20225982

Contents

1 Introduction

As multiprocessor systems become increasingly prevalent in modern computing, efficient process scheduling is essential to fully leverage their potential. Scheduling algorithms play a crucial role in optimizing processor usage, reducing latency, and ensuring that workloads are distributed evenly across multiple processors. This project undertakes the implementation and evaluation of several well-known scheduling algorithms including First-Come, First-Served (FCFS), Shortest Job First (SJF), Shortest Time-to-Completion First (STCF), Round Robin (RR), Priority Scheduling, Multilevel Feedback Queue (MLFQ) within a simulated multiprocessor environment. By focusing on key performance metrics such as throughput, average waiting time, response time, and CPU utilization, the study provides a comprehensive analysis of each algorithmâs strengths and weaknesses in multiprocessor settings.

Through detailed experiments, we assess the suitability of each algorithm for various types of workloads and discuss the specific challenges associated with multiprocessor scheduling, including load balancing, processor affinity, and the overhead of context switching. Our results indicate that algorithms like MLFQ offer adaptive and proportional CPU allocation capabilities, making them well-suited for systems with diverse process types, while simpler approaches like FCFS and RR excel in environments with lower scheduling complexity. Priority Scheduling and STCF are shown to be effective for time-critical tasks but require careful management to prevent issues like starvation and high overhead. This project highlights the trade-offs between fairness, efficiency, and computational overhead across different algorithms, offering insights that could guide the design of more sophisticated, hybrid scheduling strategies for multiprocessor systems. Finally, we suggest future directions for research, including machine learning-based adaptive scheduling methods to address dynamic workload demands in real-time systems.

2 Challenges of scheduling in multiprocessor systems

Scheduling in multi-processor systems presents several challenges that must be addressed to ensure efficient performance, resource utilization, and system stability. These challenges arise due to the need for effective task distribution, processor management, inter-process coordination, and scalability. The key challenges include:

- **Load Balancing:** Effective task distribution across processors is essential to prevent both overload and underutilization, thereby optimizing overall system performance. Achieving an equitable workload distribution is particularly challenging in heterogeneous and dynamic computing environments.
- **Processor Affinity:** Processes often execute more efficiently on processors they have previously utilized due to caching effects. However, maintaining affinity while adapting to workload fluctuations and system constraints poses a significant challenge, especially in environments with dynamic task allocation.

- **Synchronization:** Coordinating processes across multiple processors necessitates mechanisms to prevent data inconsistencies, race conditions, and deadlocks. Efficient synchronization techniques are crucial to ensuring system stability and correctness while minimizing performance overhead.
- **Scalability:** As the number of processors increases, the complexity of scheduling, load balancing, and synchronization grows exponentially. Scalable scheduling strategies are imperative to maintaining efficiency and responsiveness in large-scale computing systems.

3 First-Come, First-Served

3.1 Description

The **First-Come, First-Serve (FCFS)** scheduling algorithm for a **multiprocessor system** is a non-preemptive scheduling algorithm where processes are executed in the order they arrive without interruption until completion:

- Each processor can handle one process at a time.
- When all processors are occupied, new arriving processes are stored in a waiting queue until a processor becomes free.
- Once a process is completed, the processor marks the process as finished. Completed processes are recorded, and the system stops once all processes are completed.

3.2 Implementation

The FCFS scheduling implementation for a multiprocessor system includes several key components, each of which is detailed below:

Initialization

- `processor[4]` and `WillGoToTheQ[4]` are initialized to -1, indicating that all four processors are idle at the start and there are no processes queued to enter any specific processor.
- `finished_processes` is set to 0 to track the count of completed processes.
- The variable `i` is initialized to 0 to iterate over the processes in order of their arrival time.
- `waiting` is initialized as a queue to manage processes that are ready to execute but cannot yet be assigned to a processor due to all processors being busy.

Loop

The main scheduling loop operates over discrete time units (`t`), and each iteration performs the following actions:

- **Processor Assignment:** For each processor, if it is idle (`processor[idx_p] == -1`) and a process has arrived (`processes[i].arrive_time <= t`), that process is assigned to the processor.
- **Waiting Queue Management:** If there are additional processes that have arrived but no processors available, these processes are added to the `waiting` queue.
- **Process Execution and Completion Check:** Each assigned process runs for its burst time and is marked complete if its burst time is reached.
- The loop ends and prints results when `finished_processes` matches `size` (total number of processes).

CPU Allocation When a Process Reaches a Time Slice

- Each time unit t, the loop iterates over the four processors to check if a process has completed its burst time.
- If a process completes its current burst time, it is marked as finished, removed from the processor, and counted in `finished_processes`.
- When a processor becomes free, the process at the front of the `waiting` queue (if available) is assigned to that processor.

```
Running scheduler: FCFS
processor 1 : P1 | 0  P1 | 1  P1 | 2  P1 | 3  P1 | 4  P1 | 5  P1 | 6  P1 | 7  i | 8  i | 9  i | 10  i | 11

processor 2 : P2 | 0  P2 | 1  P2 | 2  P2 | 3  P5 | 4  P5 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  P5 | 10  i | 11

processor 3 : i | 0  P3 | 1  P3 | 2  P3 | 3  P3 | 4  P3 | 5  P3 | 6  P7 | 7  P7 | 8  P7 | 9  P7 | 10  P7 | 11

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P4 | 4  P6 | 5  P6 | 6  i | 7  i | 8  i | 9  i | 10  i | 11

----------------------processes----------------------
P1 arrived at 0 with burst time 8. It's turn arround time = 7
P2 arrived at 0 with burst time 4. It's turn arround time = 3
P3 arrived at 1 with burst time 6. It's turn arround time = 5
P4 arrived at 2 with burst time 3. It's turn arround time = 2
P5 arrived at 2 with burst time 7. It's turn arround time = 8
P6 arrived at 3 with burst time 2. It's turn arround time = 3
P7 arrived at 3 with burst time 5. It's turn arround time = 8
-----------------------------------------------------
average turn around time = 5.14286
```

3.3 Advantages and Disadvantages

Advantages

- **Simplicity**: FCFS is easy to implement and understand.
- **Predictability**: FCFS provides predictable execution by following a fixed order of arrival.

Disadvantages

- **Inefficiency**: FCFS may lead to long waiting times for processes with short burst times if they are queued after long tasks, known as the *convoy effect*.
- **Poor Load Balancing**: FCFS doesn't consider load distribution among processors, so some processors may remain idle while others are overloaded.
- **Lack of Prioritization**: Processes are not prioritized by urgency or processing time, which can lead to inefficient use of processor time.

4 Shortest Job First

4.1 Description

The **Shortest Job First (SJF)** scheduling algorithm for a **multiprocessor system** is a scheduling algorithm where processes with the shortest burst time are selected for execution first:

- Each processor can handle one process at a time.
- Processes are sorted by burst time, and a priority queue (`waiting`) is used to manage processes based on their burst time.
- Once a process is completed, the processor marks the process as finished. Completed processes are recorded, and the system stops once all processes are complete.

4.2 Implementation

The implementation of the SJF scheduling algorithm is structured as follows:

Initialization

- Arrays `processor` and `WillGoToTheQ` are initialized to -1 to indicate that each of the four processors is initially idle and the waiting queue is empty.

- A custom comparator, `CompareProcess`, is defined to prioritize processes with shorter burst times. This comparator is then used to initialize the priority queue `waiting`.
- The integer variable `finished_processes` is initialized to 0 to track the number of completed processes. The variable `i` is initialized to 0 to iterate over the list of processes.

Loop

The main scheduling loop iterates over each time unit t in the system using a `for` loop:

- The outer loop continuously increments the time unit t and checks for idle processors to assign new processes, updates the waiting queue with new arrivals, and executes assigned processes until all processes are complete.
- If the condition `finished_processes == size` is met (indicating all processes are complete), the loop terminates, and the results are printed.

CPU Allocation When a Process Reaches a Time Slice

- At each time unit t, the loop checks if any processor is idle by examining `processor[idx_p]`. If an idle processor is found, it assigns the next process (based on arrival time) to that processor.
- The `current_brust_time` and `state` of each assigned process are updated accordingly.
- If a process completes a time slice , it is pushed back to the priority queue `waiting` for reassignment based on the shortest remaining burst time.
- Once a process has fully executed , it is marked as finished, removed from the processor, and recorded in the `finished_processes` counter.

```
Running scheduler: SJF
processor 1 : P2 | 0  P2 | 1  P2 | 2  P2 | 3  P6 | 4  P6 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  P5 | 10  P5 | 11  P5 | 12

processor 2 : P1 | 0  P1 | 1  P1 | 2  P1 | 3  P1 | 4  P1 | 5  P1 | 6  P1 | 7  i | 8  i | 9  i | 10  i | 11  i | 12

processor 3 : i | 0  P3 | 1  P3 | 2  P3 | 3  P3 | 4  P3 | 5  P3 | 6  i | 7  i | 8  i | 9  i | 10  i | 11  i | 12

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P4 | 4  P7 | 5  P7 | 6  P7 | 7  P7 | 8  P7 | 9  i | 10  i | 11  i | 12

----------------------processes----------------------
P2 arrived at 0 with burst time 4. It's turn arround time = 3
P1 arrived at 0 with burst time 8. It's turn arround time = 7
P3 arrived at 1 with burst time 6. It's turn arround time = 5
P4 arrived at 2 with burst time 3. It's turn arround time = 2
P5 arrived at 2 with burst time 7. It's turn arround time = 10
P6 arrived at 3 with burst time 2. It's turn arround time = 2
P7 arrived at 3 with burst time 5. It's turn arround time = 6
-----------------------------------------------------
average turn around time = 5
```

4.3 Advantages and Disadvantages

Advantages

- **Efficiency**: SJF reduces the average waiting time by prioritizing processes with shorter burst times.
- **Better Load Balancing**: By selecting the shortest job next, SJF ensures quicker process turnovers, which can help distribute the load among processors more evenly.

Disadvantages

- **Complexity**: Using a priority queue adds complexity compared to simpler FCFS scheduling, especially when dynamically sorting by burst time.
- **Starvation**: Processes with long burst times may experience starvation if shorter tasks continuously enter the queue.

5 Shortest Time-to-Completion First

5.1 Description

The Shortest Time to Completion First (STCF) algorithm for multiple processors distributes processes across available CPUs using a priority queue, `waiting`, to manage processes waiting for execution. At each time unit t, processes that are ready are added to the `waiting` queue if no CPU is available. The process with the shortest remaining burst time is selected for each available CPU to optimize processing time and reduce wait time.

The `match_preferences()` function assists in reassigning processes, prioritizing reallocation to the CPU a process last used if that CPU is free, which helps to optimize performance.

5.2 Implementation

The `STCF` function is the main scheduling loop, distributing processes across multiple CPUs using the STCF method.

Initialization:

The function initializes the necessary arrays and variables:

- `waiting`: a priority queue of processes waiting to be executed, sorted by the shortest remaining burst time.
- `finished_processes`: counts the number of completed processes.

- `WillGoToTheQ`: an array that stores processes to be assigned to CPUs.
- `processor`: an array tracking which process is assigned to each of the CPUs.
- `cmp`: an instance of the `CompareProcess` class, used to prioritize processes based on their burst time in the `waiting` priority queue.
- `i`: a counter to track the position of the process currently being evaluated for arrival in the main loop.
- `pr`: an index variable to hold the position of the process currently assigned to a CPU.

Loop:

The main loop increments time t and performs the following steps:

1. **Adding processes to the queue:** For each process that has reached its arrival time (`arrive_time <= t`), if a CPU is available, assign the process to it. Otherwise, add it to the `waiting` queue.
2. **Processing tasks on CPUs:** Each CPU executes a single unit of time for the assigned process. When the burst time of a process ends, it is removed from the CPU:
   - If the process completes , mark it as finished.
   - If the processâs remaining burst time reaches zero, it is pushed back to the `waiting` queue if it requires further execution.
3. **Updating assignments:** The `match_preferences()` function reallocates processes to CPUs based on their current state and previously used CPUs.
4. **Recording Output:** The `output` array records the status of each CPU at each time unit, including idle states and assigned processes.
5. **Stopping condition:** When all processes are complete, the `print()` function outputs the results and the program terminates.

CPU allocation when a process reaches completion:

When a process completes its burst time , it is re-added to the `waiting` queue.

- On the next allocation, if the CPU previously used by the process is available, the algorithm prioritizes reassigning it to that CPU. If not, the process is assigned to the first available CPU among the multiple CPUs.
- This approach helps optimize performance and reduces latency when a process returns to a CPU.

```
Running scheduler: STCF
processor 1 : P2 | 0  P2 | 1  P2 | 2  P2 | 3  P1 | 4  P1 | 5  P1 | 6  P1 | 7  P1 | 8  i | 9  i | 10  i | 11

processor 2 : P1 | 0  P1 | 1  P1 | 2  P6 | 3  P6 | 4  P7 | 5  P7 | 6  P7 | 7  P7 | 8  P7 | 9  i | 10  i | 11

processor 3 : i | 0  P3 | 1  P3 | 2  P3 | 3  P3 | 4  P3 | 5  P3 | 6  i | 7  i | 8  i | 9  i | 10  i | 11

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P4 | 4  P5 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  P5 | 10  P5 | 11

----------------------processes-----------------------
P2 arrived at 0 with burst time 4. It's turn arround time = 3
P1 arrived at 0 with burst time 8. It's turn arround time = 8
P3 arrived at 1 with burst time 6. It's turn arround time = 5
P4 arrived at 2 with burst time 3. It's turn arround time = 2
P5 arrived at 2 with burst time 7. It's turn arround time = 9
P6 arrived at 3 with burst time 2. It's turn arround time = 1
P7 arrived at 3 with burst time 5. It's turn arround time = 6
------------------------------------------------------
average turn around time = 4.85714
```

match_preferences() Function

This function assigns processes from `WillGoToTheQ` to CPUs based on two strategies:

1. First loop: Reassign processes to the CPU they last used if that CPU is available.
2. Second loop: Assign remaining processes in `WillGoToTheQ` to the first available CPU found.

CompareProcess Class

This class defines the priority criteria for the processes in the `waiting` queue:

- The class compares two processes based on their `current_brust_time` and assigns a higher priority to the process with the shorter burst time.
- Used to maintain the priority ordering of processes in the `waiting` queue, ensuring that the shortest remaining burst time is always prioritized.

Output Recording

The `output` array records the process or idle state for each CPU at every time unit. For each CPU:

- If the CPU is idle, the output is set to ‘"i"‘.
- If a process is running, the process name is recorded in the `output` array.

5.3 Advantage and Disadvantage

Advantage

- **Minimized Average Wait Time**: STCF selects the process with the shortest remaining time, minimizing the average wait time and improving scheduling efficiency.
- **Reduced Latency**: Prioritizing the reassignment of processes to previously used CPUs reduces latency due to caching.

Disadvantage

- **High Resource Contention**: When many processes have significantly different burst times, some may experience longer wait times due to competition for resources.

6 Round Robin

6.1 Description

The Round Robin algorithm for multiple processors distributes processes across four CPUs by maintaining a queue, `my_queue`, to manage processes waiting for execution. At each time unit t, ready processes are assigned to any available CPU; if no CPU is available, they are added to `my_queue`. Each CPU executes a process for a fixed time interval (`time_slice`). If a process completes or reaches the end of its `time_slice`, it is removed from the CPU and placed back in the queue if it is not yet finished.

The `match_preferences()` function aids in reassigning processes, prioritizing the CPU the process last used to optimize performance.

6.2 Implementation

This function is the main scheduling loop, distributing processes across four processors using the Round Robin method.

Initialization:

The function initializes the necessary arrays and variables:

- `my_queue` - queue of processes waiting to be executed.
- `finished_processes` - counts the number of completed processes.
- `WillGoToTheQ` - array that stores processes to be assigned to processors.

- `processor` - array tracking which process is assigned to each of the four processors.
- `time_slice` - the maximum time (10 units) that each process can run before being paused and returned to the queue.

Loop:

The main loop increments time t and performs the following steps:

1. **Adding processes to the queue:** For each process that has reached time t, if there is an available processor, assign the process to it. Otherwise, add it to `my_queue`.
2. **Processing tasks on processors:** Each processor executes one time unit for the assigned process. When the `time_slice` ends or the process completes its burst time, it is removed from the processor:
   - If the process completes, mark it as finished.
   - If the `time_slice` ends, return the process to `my_queue`.
3. **Updating assignments:** The `match_preferences()` function reallocates processes to processors based on previous assignments, if available.
4. **Stopping condition:** When all processes are complete, the function calls the `print()` function to output the results and terminates.

CPU allocation when a process reaches the time slice

When a process reaches its `time_slice`, it is re-added to the queue (`my_queue`).

- On the next assignment, if the CPU that the process previously used is available, the algorithm prioritizes assigning it back to that CPU. Otherwise, it assigns the process to the first available CPU among the four CPUs.
- This approach helps optimize performance and reduces latency when a process returns to the CPU.

match_preferences() Function

This function assigns processes from `WillGoToTheQ` to processors based on two strategies:

1. First loop: Reassign processes to the CPU they last used if that CPU is available.
2. Second loop: Assign remaining processes in `WillGoToTheQ` to the first available CPU found.

```
Running scheduler: RR
processor 1 : P1 | 0  P1 | 1  P1 | 2  P1 | 3  P1 | 4  P1 | 5  P1 | 6  P1 | 7  i | 8  i | 9  i | 10  i | 11

processor 2 : P2 | 0  P2 | 1  P2 | 2  P2 | 3  P5 | 4  P5 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  P5 | 10  i | 11

processor 3 : i | 0  P3 | 1  P3 | 2  P3 | 3  P3 | 4  P3 | 5  P3 | 6  P7 | 7  P7 | 8  P7 | 9  P7 | 10  P7 | 11

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P4 | 4  P6 | 5  P6 | 6  i | 7  i | 8  i | 9  i | 10  i | 11

----------------------processes-----------------------
P1 arrived at 0 it's turn arround time = 7
P2 arrived at 0 it's turn arround time = 3
P3 arrived at 1 it's turn arround time = 5
P4 arrived at 2 it's turn arround time = 2
P5 arrived at 2 it's turn arround time = 8
P6 arrived at 3 it's turn arround time = 3
P7 arrived at 3 it's turn arround time = 8
------------------------------------------------------
average turn around time = 5.14286
```

6.3 Advantage and Disadvantage

Advantages

- **Fairness**: Each process gets an equal share of CPU time, ensuring no single process monopolizes resources and preventing starvation.
- **Simplicity**: Round Robin is straightforward to implement, using a queue and fixed time slice, making it easy to manage.
- **Good for Time-Sharing**: Especially suited for time-sharing systems, it provides predictable wait times and maintains responsiveness in interactive environments.
- **Effective for Short Tasks**: Performs efficiently for processes with similar, short durations, cycling through them quickly and evenly.

Disadvantages

- **High Context-Switching Overhead**: Frequent context switches can cause high overhead, especially with a short time slice, reducing system efficiency.
- **Not Ideal for Long Tasks**: Processes with long execution times face high latency, as they need multiple cycles to complete, delaying completion.
- **Time Slice Selection**: Selecting the optimal time slice is challenging; too short increases overhead, while too long affects responsiveness.
- **Poor for CPU-Bound Tasks**: Round Robin can decrease throughput for CPU-bound tasks, as frequent interruptions prevent extended processing.

7 Priority Scheduling

7.1 Description

This priority-based scheduling algorithm efficiently handles multiple processes across multiple CPUs by assigning them based on priority levels. Processes with higher priorities are scheduled first, ensuring balanced and fair workload management across the CPUs.

The algorithm operates through a few essential components:

- **Main Scheduling Loop**: This loop is responsible for the core scheduling operations. It controls process assignment to CPUs and manages a waiting queue where processes await their turn based on priority. Each cycle in the loop represents a unit of time, during which it updates process states and redistributes workload as needed.
- **CPU Assignment Preferences**: The algorithm tries to maintain CPU affinity by reassigning processes to the same CPUs they previously used whenever possible. This preference reduces the need for context switching and helps improve overall efficiency.

7.2 Implementation

The primary scheduling function, `Priority`, controls the scheduling loop and manages process assignment to CPUs and priority-based sorting.

Initialization:

The algorithm initializes essential arrays and variables:

- `waiting`: A priority queue storing processes waiting for CPU allocation, ordered by priority.
- `finished_processes`: Counts the number of processes that have completed.
- `processor`: An array indicating the current process assigned to each CPU.
- `WillGoToTheQ`: Temporarily stores processes ready for CPU assignment.
- `cmp`: An instance of `CompareProcesspriority` used to prioritize processes based on priority in the waiting queue.
- `i`: A counter to track processes that have arrived by the current time.
- `pr`: Tracks the current position of the process assigned to a CPU.

Loop:

The main loop increments time t and follows these steps:

1. Assigning Processes to CPUs and Waiting Queue:
   - Processes arriving at time t are directly assigned to an available CPU if one is idle, or added to the waiting queue if all CPUs are busy.
2. Updating CPU Assignments:
   - Processes assigned to CPUs are executed for one time unit, updating their `time_consumed`.
   - If the `time_consumed` reaches the `current_burst_time`, the process completes.
   - Depending on remaining phases, the process is marked as finished.
3. Updating `WillGoToTheQ`:
   - Processes from the waiting queue are transferred to `WillGoToTheQ`, ready for assignment to CPUs as they become available.
4. Executing `match_preferences()`:
   - The `match_preferences` function assigns processes from `WillGoToTheQ` to available CPUs, attempting to keep them on their previously used CPUs when possible.
5. Stopping Condition:
   - The loop ends once `finished_processes` equals the total number of processes, and the `print()` function outputs the results.

CPU Allocation for Time Slice

When a process completes its burst time , it may be reassigned to a CPU or returned to the waiting queue, ensuring efficiency and fairness:

- Burst Time Completion:
  1. When a process reaches the end of its burst time, the process is removed from the scheduling cycle.
- Assignment Preferences:
  - `match_preferences` attempts to assign processes to the same CPU they used previously, improving efficiency through processor affinity.
  - If the preferred CPU is busy, the process is allocated to the first available CPU, ensuring even workload distribution.

`CompareProcesspriority` **Class**

This class defines the comparison criteria for processes in the waiting queue:

- Processes are prioritized based on their assigned priority levels, with higher-priority processes scheduled sooner.

Output Recording

The output array records the state of each CPU at every time unit:

- An idle CPU is denoted by "i", while a busy CPU shows the `process_name` of the executing process.

```
Running scheduler: Priority
processor 1 : P1 | 0  P1 | 1  P1 | 2  P1 | 3  P1 | 4  P1 | 5  P1 | 6  P1 | 7  i | 8  i | 9  i | 10  i | 11  i | 12

processor 2 : P2 | 0  P2 | 1  P2 | 2  P2 | 3  P6 | 4  P6 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  P5 | 10  P5 | 11  P5 | 12

processor 3 : i | 0  P3 | 1  P3 | 2  P3 | 3  P3 | 4  P3 | 5  P3 | 6  i | 7  i | 8  i | 9  i | 10  i | 11  i | 12

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P4 | 4  P7 | 5  P7 | 6  P7 | 7  P7 | 8  P7 | 9  i | 10  i | 11  i | 12

----------------------processes----------------------
P1 arrived at 0 with burst time 8. It's turn arround time = 7
P2 arrived at 0 with burst time 4. It's turn arround time = 3
P3 arrived at 1 with burst time 6. It's turn arround time = 5
P4 arrived at 2 with burst time 3. It's turn arround time = 2
P5 arrived at 2 with burst time 7. It's turn arround time = 10
P6 arrived at 3 with burst time 2. It's turn arround time = 2
P7 arrived at 3 with burst time 5. It's turn arround time = 6
-----------------------------------------------------
average turn around time = 5
```

7.3 Advantages and Disadvantages

Advantages

- **Fair Priority Scheduling**: The algorithm ensures high-priority processes are scheduled promptly, providing fair CPU time distribution based on priority.
- **Efficient Resource Management**: By tracking process states and CPU usage, the algorithm balances workloads effectively across multiple processors.

Disadvantages

- **Overhead**: Frequent checks and queue manipulations increase overhead, especially with numerous processes.

8 Multilevel Feedback Queue

8.1 Description

The Multi-Level Feedback Queue (MLFQ) algorithm for process scheduling uses multiple priority queues to manage processes waiting for CPU time. At each time unit t, processes are added to the appropriate queue based on their behavior and priority level. Processes with higher priorities are given CPU time first, while lower-priority processes must wait their turn. If a process exceeds its time slice, it is moved to a lower-priority queue. Conversely, processes that don't consume their entire time slice are moved to a higher-priority queue to ensure fairness.

The `match_preferences()` function reassigns processes to CPUs, prioritizing reallocation to the CPU a process last used if that CPU is free. This helps optimize performance by minimizing unnecessary context switches. This cycle continues until all processes are completed.

8.2 Implementation

Initialization:

- `my_queue[]`: This array contains queues, where each queue holds processes with a corresponding priority level. Processes in each queue are handled in FIFO (First In, First Out) order. Higher priority queues are executed sooner.
- `finished_processes`: This variable tracks the number of processes that have been completed. The algorithm stops once the count of finished processes equals the total number of processes.
- `WillGoToTheQ[]`: This array stores processes that are about to be distributed to processors. These may include newly arrived processes or processes waiting to be rescheduled after completing a time slice .
- `processor[]`: This array tracks the state of each processor. Each element stores the index of the process assigned to that processor. An element value of `-1` indicates that the processor is idle.
- `current_queue`: This variable keeps track of the priority level of the current queue. Initially set to 0, it represents the lowest priority queue.

Main Loop:

The algorithm operates in discrete time units, incrementing the time t at each iteration. For each time unit, the following steps are performed:

1. **Add Processes to the Queue:** When a process arrives at time t, the algorithm checks for idle processors. If an idle processor is found, the process is assigned to it. If no idle processors are available, the process is added to the queue `my_queue[current_queue]` based on its priority.
2. **Process Work on Processors:** Each processor executes one time unit of the process assigned to it. After completing a time unit or reaching the end of its time slice, there are two possibilities:
   - If the process completes, the process is marked as completed.
   - If the process runs out of its time slice, it is placed back in the queue with its current priority.
3. **Update Distribution:** The `match_preferences()` function is called to redistribute processes from `WillGoToTheQ[]` to processors:
   - If the processor previously assigned to a process is idle, the process is reassigned to it.
   - If no previously used processor is available, the process is assigned to the first available idle processor.

match_preferences() Function

This function assigns processes from `WillGoToTheQ` to processors based on two strategies:

1. First loop: Reassign processes to the CPU they last used if that CPU is available.
2. Second loop: Assign remaining processes in `WillGoToTheQ` to the first available CPU found.

Handling Processes That Run Out of Time Slice

When a process either runs out of its time slice, it is returned to the queue (`my_queue[]`). If there is an idle processor that the process previously used, the algorithm prioritizes reassigning the process to that processor. If no such processor is available, the process is assigned to the first idle processor found in the `processor[]` array. [1] [2]

```
Running scheduler: MLFQ
processor 1 : P1 | 0  P1 | 1  P5 | 2  P5 | 3  P3 | 4  P3 | 5  P3 | 6  P3 | 7  i | 8  i | 9  i | 10

processor 2 : P2 | 0  P2 | 1  P2 | 2  P6 | 3  P6 | 4  P5 | 5  P5 | 6  P5 | 7  P5 | 8  P5 | 9  i | 10

processor 3 : i | 0  P3 | 1  P3 | 2  P7 | 3  P7 | 4  P1 | 5  P1 | 6  P1 | 7  P1 | 8  P1 | 9  P1 | 10

processor 4 : i | 0  i | 1  P4 | 2  P4 | 3  P2 | 4  P4 | 5  P7 | 6  P7 | 7  P7 | 8  i | 9  i | 10

----------------------processes-----------------------
P1 arrived at 0 with burst time 8. It's turn arround time = 10
P2 arrived at 0 with burst time 4. It's turn arround time = 4
P3 arrived at 1 with burst time 6. It's turn arround time = 6
P4 arrived at 2 with burst time 3. It's turn arround time = 3
P5 arrived at 2 with burst time 7. It's turn arround time = 7
P6 arrived at 3 with burst time 2. It's turn arround time = 1
P7 arrived at 3 with burst time 5. It's turn arround time = 5
------------------------------------------------------
average turn around time = 5.14286
```

8.3 Advantage and Disadvantage

Advantage

- **Optimizes performance:** Processes with higher priority are executed first, helping to reduce latency for important processes.
- **Real-time support:** The algorithm helps efficiently distribute CPU resources in systems with many processes and real-time requirements.
- **Ability to move between priority levels:** Processes can move from a lower priority level to a higher one if they do not use up their time slice, optimizing resource distribution.

Disadvantage

- **Complex Implementation:** The management of multiple priority queues and dynamic priority adjustments can make MLFQ difficult to implement and maintain.
- **Starvation of Low-Priority Processes:** Low-priority processes may be starved if high-priority processes continuously consume CPU time.
- **Context Switching Overhead:** Frequent context switches between queues due to process priority adjustments can lead to increased overhead and reduced efficiency.

9 Comparison of Scheduling Algorithm

The below table would represent the execution time of each scheduling algorithm in a large-scale number of processes:

Algorithm	**Average Round Time (s)**
First-Come, First-Served	20.65
Shortest Job First	14.45
Shortest Time-to-Completion First	14.175
Round Robin	21.5
Priority Scheduling	18.475
Multilevel Feedback Queue	27.625

- In environments where predicting process burst times is infeasible, **Round Robin (RR)** and **Multilevel Feedback Queue (MLFQ)** are more suitable due to their flexibility. RR ensures fair CPU allocation by rotating processes in fixed time slices, making it ideal for multitasking. MLFQ dynamically adjusts process priorities based on execution behavior, allowing the system to handle diverse workloads without prior burst time knowledge.
- **Shortest Job First (SJF)** and **Shortest Time-to-Completion First (STCF)** require the system to **predict** burst times, which is often challenging in real-world scenarios.

The following table presents an overview of different CPU scheduling algorithms and their suitability for multi-processor systems. It highlights how each algorithm performs in such environments, considering factors like flexibility, efficiency, and task distribution.

Algorithm	**Suitability in Multi-processor Systems**
FCFS	Poor, lacks flexibility.
SJF	Good if proper prediction.
STCF	Effective with low load.
RR	Fair but less efficient.
Priority	Ideal for real-time systems.
MLFQ	Well-suited for multiprocessor systems, as multiple queues can help distribute tasks.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Cpu scheduling: Multi-level feedback queue." https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf. Accessed: 2024-12-31.

[2] "Multiple processor scheduling in operating system." https://www.geeksforgeeks.org/multiple-processor-scheduling-in-operating-system/. Accessed: 2024-12-31.