OPERATING SYSTEM

PROJECT WORK

**Submitted to : Dr. Mehreen Sirshar**

**Group members:**

* ABIHA NADEEM (BSE-01-3A)
* BUSHRA ASHRAF BHATTI (BSE-015-3A)

**Date:11/10/2024**

**CPU scheduling Algorithms;**

“Research paper”

**An In-Depth Performance and Scalability Analysis of Modern CPU Scheduling Algorithms in Multicore Systems: Investigating the Trade-offs between Fairness, Efficiency, and Real-time Responsiveness**.

**Fatima Jinnah Women university, Rawalpindi**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Abstract:**

The rapid evolution of multicore processors has led to new challenges in CPU scheduling, where balancing fairness, efficiency, and real-time responsiveness is essential for optimizing system performance. This paper conducts an in-depth performance and scalability analysis of key CPU scheduling algorithms in multicore environments. Using a multicore experimental setup, we evaluate each algorithm across three primary metrics fairness, measured by equitable CPU time allocation; efficiency, evaluated through CPU utilization, throughput, and energy consumption; and real-time responsiveness, assessed by latency and response times for time-sensitive tasks. Scheduling in multicore environments is more complex than in single-core systems due to the need to manage cache coherence, minimize the cost of context switches, and ensure that shared resources are used optimally. Effective scheduling maximizes resource utilization and reduces system latency, which is particularly important in systems with mixed workloads that include both real-time and non-real-time tasks.

The rise of multicore processors has revolutionized computing by enabling unparalleled levels of parallel processing, allowing multiple tasks to run simultaneously on separate cores. This leap has unlocked higher performance in a wide range of applications, from everyday computing to complex scientific simulations. However, effectively managing CPU scheduling across multiple cores presents complex challenges, especially in balancing critical factors: fairness, efficiency, and real-time responsiveness.

As more cores are added, the need for efficient scheduling becomes paramount to ensure that system resources are utilized to their fullest potential without creating bottlenecks. Fairness, for instance, demands that each process gets a reasonable share of CPU time, which can be difficult to maintain in multicore systems due to shared resources such as cache and memory. Ensuring fairness is vital to prevent certain tasks from monopolizing processing power, which could lead to system inefficiency and negatively impact user experience.

**Introduction:**

The advent of multicore processors has transformed modern computing, enabling systems to execute multiple tasks in parallel and thus dramatically improving performance, energy efficiency, and scalability. With these advancements, multicore architectures have become integral to a wide range of applications, from personal computing and data centers to real-time embedded systems. However, effectively managing task execution across multiple cores remains a complex challenge. CPU scheduling, the mechanism that allocates processor time to various tasks, is central to achieving optimal performance in multicore systems. In this context, the choice of scheduling algorithm significantly impacts system behavior, particularly in balancing critical metrics like fairness, efficiency, and real-time responsiveness.

* **Challenges in Multicore CPU Scheduling**  
  In multicore systems, scheduling faces unique challenges due to the need to distribute tasks equitably across cores (fairness), maximize CPU utilization (efficiency), and meet latency requirements for time-sensitive applications (real-time responsiveness). Each scheduling algorithm offers different advantages and limitations in addressing these criteria. Fairness ensures that all tasks receive adequate access to processing resources, efficiency focuses on maximizing throughput and minimizing idle time, and real-time responsiveness prioritizes rapid handling of high-priority or time-sensitive tasks. Striking a balance among these goals is essential, as prioritizing one often leads to trade-offs in the others.

**Research Question:**  
This study aims to analyze and compare the performance and scalability of various modern CPU scheduling algorithms in multicore environments, focusing on their effectiveness in balancing fairness, efficiency, and real-time responsiveness. Specifically, we investigate the trade-offs associated with the Completely Fair Scheduler (CFS), Round Robin, Priority Scheduling, and long job first(LJF), etc. By evaluating these algorithms in a controlled multicore setting, we seek to provide insights into their relative strengths and weaknesses and to identify the conditions under which each is most suitable.

**Outline of the paper:**

This paper is organized as follows:

1. First, we review about Multicore system and multicore processor, its working, advantages and disadvantages, impact, applications and architecture.
2. Then we study about what is CPU scheduling, its scheduling criteria’s in operating system.
3. Types of CPU scheduling.
4. Then we discussed about different CPU algorithm used to in hence the performances, scalability, it’s trade-offs between fairness, efficiency, and real-time responsiveness.

**Literature review:**

Efficient CPU scheduling in multicore systems is crucial for optimizing performance, fairness, and real-time responsiveness. As multicore processors have become standard, scheduling algorithms must distribute tasks effectively across multiple cores, balancing fairness, efficiency, and real-time constraints.

**CPU Scheduling in Multicore Systems**

Multicore systems use global, partitioned, and hybrid scheduling approaches to manage tasks. Global scheduling assigns tasks to any available core, while partitioned scheduling assigns tasks to fixed cores. Hybrid approaches aim to combine the benefits of both. Research indicates that while global scheduling offers flexibility, it can incur high overhead, while partitioned methods scale better but may limit flexibility.

**Fairness vs. Efficiency**

Fairness ensures equal CPU time for tasks, while efficiency seeks to maximize throughput. Fair Queueing and the Completely Fair Scheduler (CFS) have been applied to achieve fairness, though they can reduce efficiency in some cases. On the other hand, efficiency-focused algorithms, such as load balancing and Backfilling, aim to maximize resource usage but may sacrifice fairness.

**Real-Time Responsiveness**

In real-time systems, meeting deadlines is crucial. Algorithms like Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS) are adapted for multicore systems but face challenges in scalability due to task migration overhead. Hybrid models combining global and real-time scheduling offer better responsiveness, though they may reduce scalability.

**Scalability**

As multicore systems grow, scalability becomes a key concern. Dynamic Load Balancing and Work Stealing are used to improve scalability by redistributing tasks across cores, but they introduce synchronization overhead that can affect performance. Balancing scalability with fairness and efficiency remains a challenge.

**Emerging Approaches**

Recent trends include the use of machine learning for adaptive scheduling and energy-aware algorithms to optimize both performance and sustainability. Additionally, heterogeneous architectures, including CPUs and accelerators like GPUs, require new scheduling paradigms to manage diverse resources effectively.

**Conclusion**

The literature shows that modern CPU scheduling algorithms must balance fairness, efficiency, and real-time responsiveness in multicore systems. While traditional methods continue to evolve, emerging techniques like machine learning and energy-aware scheduling offer promising solutions for meeting the complex demands of contemporary computing environments.

**Methodology :**

This section outlines the approach taken to conduct an in-depth analysis of modern CPU scheduling algorithms in multicore systems, focusing on fairness, efficiency, and real-time responsiveness. The methodology includes the following steps:

**1. Research Approach:**

* The study adopts a theoretical and analytical framework, relying on an extensive review of existing literature.
* A comparative approach is used to examine how various algorithms perform under the challenges of multicore environments.

**2. Data Collection and Review**

* **Sources of Information**: Peer-reviewed articles, conference proceedings, technical documentation, and books related to multicore scheduling.
* **Search Strategy**: Keywords like *“multicore scheduling,”* *“fairness in scheduling,”* *“real-time CPU responsiveness,”* and *“scalability of scheduling algorithms”* were used in databases such as IEEE Xplore, SpringerLink, and ACM Digital Library.
* **Selection Criteria**: Sources were included based on relevance, recency, and focus on the performance metrics being analyzed.

**3. Analytical Framework**

* **Algorithms Selection**: The study examines key CPU scheduling algorithms, including:
* **Traditional Algorithms**: Round Robin, Priority Scheduling.
* **Fairness-Oriented Algorithms**: Completely Fair Scheduler (CFS).
* **Real-Time Algorithms**: Earliest Deadline First (EDF), Rate Monotonic Scheduling (RMS).
* **Performance Metrics**:
* **Fairness**: Evaluated based on equitable CPU time allocation.
* **Efficiency**: Measured through throughput, CPU utilization, and energy efficiency.
* **Real-Time Responsiveness**: Assessed using metrics like latency and deadline adherence.

**4. Categorization of Challenges**

* **Multicore-Specific Issues**: The study explores challenges like cache coherence, task migration overhead, and shared resource contention.
* **Trade-offs**: A detailed analysis of the trade-offs between fairness, efficiency, and real-time responsiveness in multicore scheduling.

### 5. Challenges in Multicore Scheduling

* The study delves into issues such as:
* Task migration and load balancing.
* Synchronization overhead in shared memory environments.
* Energy efficiency trade-offs in multicore systems.
* These challenges were analyzed by comparing the theoretical solutions proposed in various studies.

### 6. Tools and Techniques for Analysis

Although no experiments were conducted, the study utilized:

* **Conceptual Comparisons**: Juxtaposing algorithms against each other based on theoretical models.
* **Frameworks and Models**: Referencing models for evaluating fairness (e.g., Jain’s Fairness Index), efficiency (e.g., CPU utilization), and responsiveness (e.g., task latency).
* **Visualization Techniques**: Graphical representation of trade-offs to enhance understanding.

### 7. Limitations

* This study is limited to theoretical insights derived from secondary data and does not include experimental validation.
* Hardware-specific behaviors, such as architecture-dependent performance, are beyond the scope of this research.

**Multicore system:**

A multicore system refers to a computer architecture that includes multiple processing units, known as cores, on a single chip. Each core is capable of executing its own instructions independently of the others, which allows for parallel processing. This means that a multicore processor can handle more tasks simultaneously, improving performance, especially for tasks that can be divided into smaller, parallelized parts.

Key features of multicore systems:

1. **Multiple Cores**: A multicore processor typically has two or more cores, with modern systems having four, eight, or more cores.
2. **Parallel Processing**: Cores can execute different instructions at the same time, making the system more efficient for multi-threaded applications (like gaming, video editing, and scientific computing).
3. **Improved Performance**: Multicore systems can improve performance for tasks that can be divided into independent threads, as each core works on a separate thread simultaneously.
4. **Energy Efficiency**: By using multiple smaller cores instead of a single powerful core, multicore systems can be more energy-efficient while still providing the required performance.

In general, multicore systems are widely used in modern computing devices, such as desktops, laptops, smartphones, and servers, to enhance performance and support the demands of multitasking and resource-intensive applications.

### Multicore Processor:

A multicore processor is a single computing unit that contains two or more independent central processing units (CPUs), known as "cores," on a single integrated circuit (IC) or chip. Each core functions as a separate processing unit capable of executing tasks independently or collaboratively with other cores.

This design enables simultaneous execution of multiple instructions, significantly boosting processing power, efficiency, and multitasking capabilities. Multicore processors are prevalent in modern devices such as smartphones, laptops, servers, and embedded systems, where computational demands are high and efficiency is crucial.

### Evolution of Processors: From Single-Core to Multicore:

#### Single-Core Processor

* **Definition**: Early processors contained a single CPU core that could execute one instruction stream at a time.
* **Examples**: Intel 8086, AMD K5.
* **Limitations**:
* **Performance Bottlenecks**: Limited ability to handle multiple tasks simultaneously.
* **Clock Speed Limits**: As demands increased, manufacturers attempted to boost performance by increasing clock speeds. However, higher clock speeds led to heat generation and energy inefficiencies.

### Multicore Processor

A **multicore processor** is a single computing component that integrates two or more independent central processing units (CPUs) called "cores" on a single chip. Each core can independently execute instructions, enabling parallel processing. This design improves the performance and efficiency of the processor by allowing multiple tasks to be executed simultaneously.

### Multicore Processor Work?

Multicore processors work by dividing tasks into smaller sub-tasks and assigning them to different cores. This is known as **parallel processing**. The processor uses advanced algorithms to manage and distribute workloads among the cores, enabling efficient execution. Multicore processors rely on:

* **Threading:** Multiple threads can run in parallel.
* **Shared Memory Access:** Cores can share access to cache memory and main memory.
* **Coordination Mechanisms:** Use of scheduling algorithms and inter-core communication to optimize task distribution.

**Advantages of Using a Multicore Processor:**

1. **Improved Performance:** Tasks are executed faster due to parallel processing.
2. **Energy Efficiency:** Multicore designs typically use less power than multiple single-core processors for the same performance.
3. **Better Multitasking:** Handles multiple applications or processes simultaneously.
4. **Scalability:** Ideal for applications designed for parallel execution, such as video rendering and simulations.

### Disadvantages of Using a Multicore Processor:

1. **Software Dependency:** Applications must be optimized for multithreading to benefit fully.
2. **Increased Complexity:** Design, debugging, and optimization are more complex than single-core systems.
3. **Heat Generation:** More cores can lead to higher temperatures, requiring better cooling solutions.
4. **Cost:** Multicore processors are generally more expensive than single-core ones.

#### Transition to Multithreading:

* **Introduction of Hyper-Threading**: Techniques like simultaneous multithreading (e.g., Intel's Hyper-Threading) allowed single-core processors to handle multiple instruction threads to simulate multitasking.
* **Limitations**: This approach improved efficiency but couldn't fully overcome physical and performance constraints.

#### Emergence of Multicore Processors:

* **Reasons for Transition**:
* **Heat and Power Management**: Increasing clock speeds in single-core processors became impractical due to excessive heat and energy consumption.
* **Demand for Parallelism**: Applications like gaming, multimedia, and scientific simulations required real-time processing of multiple tasks.
* **First Multicore Processors**: In the mid-2000s, processors like Intel’s Core Duo and AMD’s Athlon 64 X2 marked the shift to dual-core architectures.
* **Advancements**:
  + From dual-core to quad-core and octa-core designs.
  + Innovations like heterogeneous multicore processors (e.g., ARM’s big.LITTLE architecture).

### How Does a Multicore Processor Impact Server Performance?

1. **High Throughput:** Servers can handle more simultaneous requests and run complex calculations faster.
2. **Enhanced Scalability:** Applications with high parallelism scale better on multicore processors.
3. **Energy Savings:** For data centers, multicore processors provide better performance-per-watt.
4. **Load Balancing:** Multicore systems allow better distribution of workloads, improving reliability and uptime.

### Common Applications of Multicore Processors:

1. **Gaming:** Enhances graphics rendering and supports high frame rates.
2. **Video Editing and Rendering:** Handles multiple threads to process videos faster.
3. **Artificial Intelligence and Machine Learning:** Powers deep learning models and real-time inference.
4. **Servers and Cloud Computing:** Supports virtualization and high-demand web services.
5. **Mobile Devices:** Provides power efficiency and multitasking capabilities.

### Architecture of Multicore Processors:

1. **Cores:** Individual processing units capable of independent execution.
2. **Cache Memory:** Hierarchical cache (L1, L2, L3) to store frequently accessed data.
3. **Interconnects:** Communication pathways between cores and memory controllers.
4. **Memory Controller:** Manages access to shared memory.
5. **Bus Systems:** High-speed communication between cores and peripheral devices.

### Examples of Multicore Processors:

1. **Intel Core Series:**

* Intel Core i9-12900K: Features 16 cores with a hybrid architecture combining performance and efficiency cores.
* Target: Gaming, productivity, and high-performance computing.

1. **AMD Ryzen Series:**

* AMD Ryzen 9 7950X: Features 16 cores and 32 threads.
* Target: Creative professionals and gamers.

1. **Apple M1/M2:**
   * Features a mix of high-performance and high-efficiency cores.
   * Target: Laptops and desktops for professional and casual use.
2. **Qualcomm Snapdragon Series:**

* Designed for mobile devices.
* Features multiple cores optimized for AI, graphics, and energy efficiency.

1. **IBM POWER Processors:**
   * POWER10: Features advanced parallelism for data centers and enterprise computing.
2. **ARM Cortex Series:**
   * ARM Cortex-A78: Popular in smartphones, emphasizing performance and efficiency.

### Importance of Multicore Processors in Modern Computing:

#### 1. Enhanced Performance through Parallel Processing

Multicore processors excel in dividing complex workloads into smaller tasks distributed across cores. This parallel processing reduces execution time and increases efficiency, particularly for applications like video rendering, gaming, and scientific simulations.

#### 2. Energy Efficiency

Multicore designs allow tasks to be distributed among cores running at lower frequencies. This reduces power consumption compared to a single core operating at higher frequencies, making them ideal for mobile devices and energy-conscious data centers.

#### 3. Improved Multitasking

With multiple cores, a system can execute numerous processes simultaneously. For example, a smartphone can handle video playback, browsing, and background app updates without noticeable slowdowns.

#### 4. Real-Time Applications

Modern applications like artificial intelligence (AI), virtual reality (VR), and autonomous vehicles require high computational power and responsiveness. Multicore processors ensure seamless real-time processing in these scenarios.

#### 5. Scalability for Future Demands

As computing needs grow, multicore processors offer scalability by adding more cores rather than increasing clock speeds. This approach ensures sustainable performance growth while addressing energy and heat constraints.

### CPU Scheduling in Operating Systems

### Basic concept:

CPU Scheduling is the method by which an operating system decides which process (task or program) should be executed by the CPU at any given time. The goal is to optimize CPU usage, maximize system performance, and ensure fair process execution. In modern systems, CPU scheduling is crucial for managing multitasking and providing a responsive user experience.

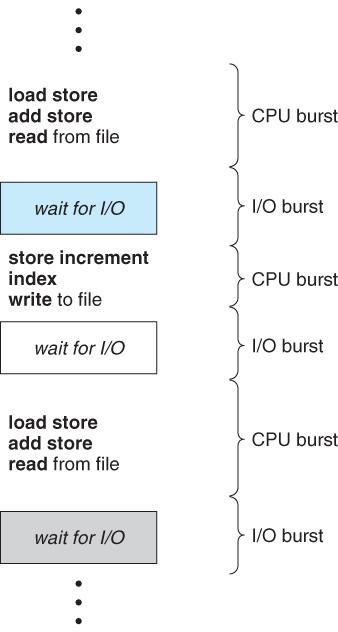
### What is the Need for a CPU Scheduling Algorithm?

The primary goals of a CPU scheduling algorithm are to:

1. **Maximize CPU Utilization:** Keep the CPU busy by selecting the next process that needs execution.
2. **Optimize Throughput:** Complete processes as efficiently as possible, maximizing the number of processes completed in a given time.
3. **Minimize Waiting Time:** Reduce the time a process spends in the ready queue before execution.
4. **Minimize Turnaround Time:** Reduce the total time from submission to completion of a process.
5. **Fairness:** Ensure that all processes get a fair share of the CPU, preventing starvation of low-priority tasks.
6. **Improve Response Time:** Ensure quick response times for interactive processes.

Without an efficient scheduling algorithm, the system might experience delays, inefficiencies, or unfair resource allocation.

**Alternating sequence of CPU and iI/o burst:**

****

### Terminologies Used in CPU Scheduling( scheduling criteria’s):

1. **Process:** A program in execution. Each process has its own process control block (PCB).
2. **CPU Burst:** The time a process needs to use the CPU to execute its instructions.
3. **I/O Burst:** The time a process spends waiting for I/O operations (e.g., disk read/write).
4. **Ready Queue:** A queue that holds all processes that are ready to execute but waiting for CPU time.
5. **Waiting Queue:** A queue that holds processes waiting for I/O operations to complete.
6. **Context Switching:** The process of saving the state of the currently running process and loading the state of the next process to execute.
7. **Turnaround Time:** The total time taken from the arrival of the process to its completion (includes both CPU and I/O time).
8. **Waiting Time:** The total time a process spends in the ready queue waiting for CPU access.
9. **Response Time:** The time it takes for the system to respond to a request, usually important for interactive systems.
10. **Throughput:** The number of processes completed in a given time period.

### Designing (Structure) a CPU Scheduling Algorithm:

Designing a CPU scheduling algorithm involves:

1. **Defining Criteria:** The goal of the algorithm is to minimize waiting time, response time, and maximize throughput.
2. **Input Information:** The algorithm must have access to process priorities, burst times (CPU and I/O), and the arrival time of processes.
3. **Queue Management:** Efficiently managing ready and waiting queues, ensuring that processes are appropriately scheduled based on the algorithm’s rules.
4. **Context Switching:** Managing context switching efficiently to minimize overhead.
5. **Fairness Considerations:** Ensuring no process is indefinitely delayed or starved, especially in systems with high priority processes.

The algorithm must balance these factors depending on the system's goals, whether it's fairness, efficiency, or responsiveness.

**CPU scheduler :**

* The CPU Scheduler, also called the Short-Term Scheduler, manages how processes share the CPU in an operating system.
* It selects processes from the ready queue (a list of processes waiting for CPU time) for execution.
* Ensures efficient CPU utilization and system performance.
* Uses scheduling algorithms to decide the order of process execution (e.g., FCFS, SJF, Priority, Round Robin).
* Handles context switching, which involves saving the state of one process and restoring the state of the next.
* Balances objectives like fairness, minimizing waiting time, maximizing throughput, and ensuring responsiveness.

**Dispatcher:**

* The Dispatcher is a component of the operating system that handles the actual transfer of control of the CPU from one process to another.
* It works after the CPU Scheduler selects a process for execution.
* Performs context switching, which saves the state of the current process and loads the state of the next process.
* Switches the CPU to user mode to begin or resume the selected process.
* Ensures proper program execution by handling the transition between processes.
* Aims to minimize dispatch latency, which is the time taken by the dispatcher to complete the switching process.

### Different Types of CPU Scheduling Algorithms

CPU scheduling algorithms are categorized into **preemptive** and **non-preemptive** types based on whether the operating system can interrupt a running process to assign CPU time to another process.

### Preemptive CPU Scheduling Algorithms:

In **preemptive scheduling**, the operating system can interrupt the currently running process to allocate CPU time to another process. This is essential for systems with high interactivity, where response time is critical.

1. **Round Robin (RR):**
   * **Description:** Each process is assigned a fixed time slice (or quantum) in which it can execute. When the quantum expires, the process is interrupted, and the next process in the ready queue is given the CPU.
   * **Advantages:** Simple, fair, and responsive, especially for time-sharing systems.
   * **Disadvantages:** It may lead to high turnaround times for short processes, as long processes may monopolize the CPU for multiple time slices.
2. **Shortest Remaining Time First (SRTF):**

* **Description:** A preemptive version of Shortest Job Next (SJN). The process with the shortest remaining CPU burst time is given priority for execution.
* **Advantages:** Minimizes average waiting and turnaround time.
* **Disadvantages:** Processes with long burst times may experience starvation if short processes keep arriving.

1. **Priority Scheduling (Preemptive):**
   * **Description:** Processes are assigned a priority, and the process with the highest priority is selected to run. If a new process arrives with a higher priority than the current process, it preempts the current process.
   * **Advantages:** Useful in systems requiring strict prioritization (e.g., real-time systems).
   * **Disadvantages:** Starvation of lower-priority processes is a risk.
2. **Multilevel Queue Scheduling:**
   * **Description:** Processes are divided into different priority queues, and each queue uses a different scheduling algorithm. For example, a foreground queue could use round robin, while a background queue could use priority scheduling.
   * **Advantages:** Organizes processes based on their priority and required resources.
   * **Disadvantages:** Complex to manage and configure.

### Non-Preemptive CPU Scheduling Algorithms:

In **non-preemptive scheduling**, once a process starts executing, it is allowed to run to completion or until it voluntarily releases the CPU (for example, when it finishes its time quantum or waits for I/O).

1. **First-Come, First-Served (FCFS):**

* **Description:** The process that arrives first is executed first. This is the simplest scheduling algorithm.
* **Advantages:** Easy to implement.
* **Disadvantages:** Can cause long waiting times (convoy effect), especially when short processes arrive after long ones.

1. **Shortest Job Next (SJN) or Shortest Job First (SJF):**
   * **Description:** The process with the shortest estimated CPU burst time is executed first. This minimizes average waiting time.
   * **Advantages:** Minimizes average waiting time and is optimal if the burst time is known in advance.
   * **Disadvantages:** Hard to predict burst time and can lead to starvation for long processes.
2. **Priority Scheduling (Non-Preemptive):**
   * **Description:** Similar to the preemptive version, but once a process starts executing, it continues until completion.
   * **Advantages:** Simple and easy to implement.
   * **Disadvantages:** Starvation of low-priority processes is a risk.

**Comparison Between Preemptive and Non-Preemptive Scheduling:**

|  |  |  |
| --- | --- | --- |
| **Features** | **Preemptive Scheduling** | **Non-Preemptive Scheduling** |
| **Definition** | The operating system can interrupt a running process to allocate CPU time to another process. | Once a process starts executing, it runs to completion or until it voluntarily releases the CPU. |
| **Context switching** | Frequent context switches due to process preemption. | Less frequent context switching since processes run to completion. |
| **Cpu utilization** | Typically higher CPU utilization since the CPU is always performing tasks while waiting for others to complete. | May result in lower CPU utilization if a process blocks the CPU for long periods. |
| **Efficiency** | Can result in better CPU utilization by giving CPU time to processes that need it immediately. | Might cause CPU underutilization if long processes are running. |
| **Througput** | Might reduce throughput if frequent context switching leads to high overhead | Usually results in higher throughput because processes run to completion. |
| **Interrupting Process** | **yes** | **No** |
| **Response Time** | Faster for interactive systems | Slower for interactive systems |
| **Fairness** | More fair (processes are interrupted) | Can lead to starvation of low-priority processes |
| **Complexity** | |  | | --- | | Higher (due to frequent context switching) | | **Simpler** |
| **Types of algorithm** | Round Robin (RR), Shortest Remaining Time First (SRTF), Preemptive Priority Scheduling, Multilevel Queue Scheduling | First-Come, First-Served (FCFS), Shortest Job Next (SJN), Non-Preemptive Priority Scheduling |
| **Use case** | Suitable for time-sharing systems, real-time applications, or when fairness and responsiveness are important. | Suitable for batch systems or environments where tasks can be predicted to take a fixed time to complete. |
| **Turnaround time** | Can result in higher turnaround time if short processes are delayed by long ones (e.g., in RR). | Typically has lower turnaround time because processes run to completion once started. |
| **Example** | Round Robin, Preemptive Priority Scheduling, SRTF, Multilevel Queue Scheduling. | FCFS, Non-Preemptive Priority Scheduling, SJN. |

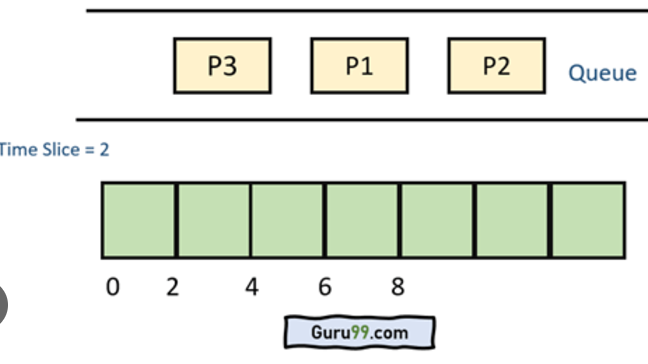
Each algorithm has its strengths and weaknesses depending on the system's goals (fairness, responsiveness, or throughput), and often hybrid approaches are used in modern operating systems.

**CPU scheduling algorithms**

**1. Multicore Round Robin (RR):**

Explanation:

**Round Robin (RR)** is a preemptive scheduling algorithm that allocates a fixed time quantum to each process. In a multicore system, RR can be adapted by assigning processes to available cores in a cyclic manner, ensuring that each core executes processes in a round-robin fashion. When a process exhausts its time quantum, it is preempted and scheduled for execution on another core or returned to the ready queue.

Advantages:

* **Fair allocation**: Each process gets an equal share of CPU time, making it fair for all processes.
* **Simple and predictable**: Easy to implement and understand; works well for time-sharing systems.
* **Preemptive**: Can handle interactive tasks effectively by giving them frequent CPU time slices.

Disadvantages:

* **Context switching overhead**: Frequent preemption can incur high overhead, especially if the time quantum is small.
* **Not optimized for heavy computational tasks**: Multithreaded or CPU-bound tasks may suffer from high context switching overhead.
* **Limited real-time responsiveness**: Round Robin may not handle real-time tasks well, as the next process may not be immediately available if another process is executing.

Characteristics:

* **Preemptive**: Tasks are preempted after a fixed time quantum.
* **Fairness**: Every process gets an equal share of the CPU time.
* **Simple**: Easy to implement but does not consider process priority or burst time.

Trade-offs:

* **Fairness**: Excellent fairness because every process gets equal CPU time.
* **Efficiency**: Can be inefficient for CPU-bound tasks due to frequent context switching.
* **Real-time Responsiveness**: Poor real-time responsiveness, as processes might be delayed by the time quantum, impacting time-sensitive tasks.

**2. Multicore First-Come, First-Served (FCFS)**

Explanation:

**FCFS** is a non-preemptive algorithm where processes are executed in the order they arrive in the ready queue. In a multicore system, processes are assigned to cores in the order they arrive, and each core handles its assigned processes using the FCFS algorithm.

Advantages:

* **Simplicity**: Easy to implement and does not require complex calculations.
* **Deterministic**: Processes are executed in the order they arrive, making the system predictable.

Disadvantages:

* **Convoy effect**: Long processes can delay the execution of shorter processes, leading to high waiting times.
* **Inefficiency for interactive tasks**: Long-running processes dominate the CPU, causing poor responsiveness for interactive tasks.
* **No preemption**: Does not handle interruptions effectively, making it unsuitable for real-time systems.

Characteristics:

* **Non-preemptive**: Once a process starts, it runs to completion.
* **Simple**: Processes are executed in the order they arrive.
* **Inefficient for mixed workloads**: Works poorly when there is a mix of short and long processes.

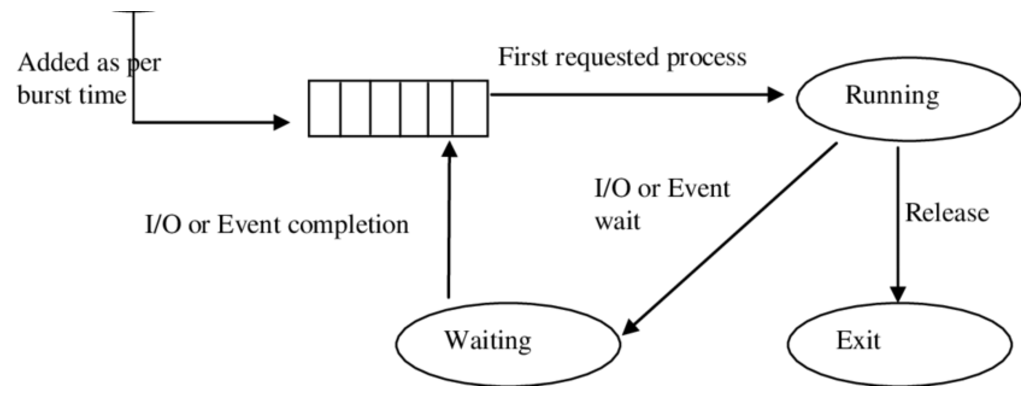
Trade-offs:

* **Fairness**: Fair in the sense that processes are executed in their arrival order, but it can lead to starvation for shorter processes.
* **Efficiency**: Can be inefficient, especially in systems with a mixture of short and long tasks.
* **Real-time Responsiveness**: Poor for real-time systems, as long tasks can delay time-critical processes.

**3. Multicore Shortest Job First (SJF)**

Explanation:

**SJF** is a non-preemptive algorithm that selects the process with the shortest burst time to execute next. In multicore systems, each core executes the shortest job from its local queue of processes. This minimizes waiting time and improves throughput, but requires knowing the burst times in advance.



Advantages:

* **Minimizes waiting time**: By executing shorter processes first, SJF minimizes the average waiting time for all processes.
* **Efficient for CPU-bound tasks**: Performs well when most tasks are of similar size or predictable burst times.

Disadvantages:

* **Starvation**: Long processes may never get executed if shorter processes keep arriving.
* **Requires knowledge of burst time**: In many systems, estimating burst time is difficult and can lead to inaccurate scheduling.
* **Not suited for real-time systems**: Hard to predict burst times, and long tasks may starve.

Characteristics:

* **Non-preemptive**: Once a process starts, it runs to completion.
* **Optimal for average waiting time**: Minimizes the average waiting and turnaround time.
* **Starvation**: Can cause starvation of longer tasks.

Trade-offs:

* **Fairness**: Not fair to long processes, which may starve if there are many short tasks.
* **Efficiency**: High efficiency for systems with tasks of predictable or similar burst times.
* **Real-time Responsiveness**: Poor real-time responsiveness, as tasks with long burst times may be delayed indefinitely.

**4. Multicore Shortest Remaining Time First (SRTF)**

Explanation:

**SRTF** is a preemptive version of SJF, where the process with the shortest remaining burst time is executed next. In a multicore system, each core selects the process with the shortest remaining time for execution. If a new process arrives with a shorter remaining time, it preempts the current process.

Advantages:

* **Minimizes waiting time**: Like SJF, it minimizes average waiting time by prioritizing shorter tasks.
* **Preemptive**: Allows real-time task handling by preempting longer tasks when a shorter task arrives.

Disadvantages:

* **Starvation**: Long-running processes may be starved indefinitely if shorter tasks keep arriving.
* **Requires knowledge of burst times**: Estimating the remaining burst time can be difficult, especially for interactive or dynamic workloads.
* **Overhead from preemption**: Constantly checking the remaining time and preempting processes introduces overhead.

Characteristics:

* **Preemptive**: Processes can be interrupted if a new process with a shorter remaining time arrives.
* **Optimal for minimizing waiting time**: Like SJF, but with preemption to handle dynamic workloads.
* **Starvation**: Long tasks may be continuously preempted by shorter tasks.

Trade-offs:

* **Fairness**: Can lead to unfairness by starving longer tasks.
* **Efficiency**: High efficiency for short tasks, but preemption overhead can reduce efficiency in some cases.
* **Real-time Responsiveness**: Better than non-preemptive algorithms like FCFS and SJF for real-time systems, as it allows for immediate execution of urgent tasks.

**5. Multicore Longest Remaining Time First (LRTF)**

Explanation:

**LRTF** is the opposite of SRTF, where the process with the **longest remaining burst time** is executed next. In a multicore system, this algorithm schedules processes with the longest remaining burst time on each core, prioritizing lengthy processes.

Advantages:

* **Prevents starvation of long tasks**: Unlike SJF or SRTF, long tasks are not starved but executed first.
* **May improve performance for certain batch tasks**: If most tasks are long-running or part of a batch process, this can optimize execution.

Disadvantages:

* **Starvation of short tasks**: Short tasks may never be executed if long tasks keep arriving.
* **Inefficient in mixed workloads**: Reduces throughput in systems with many short tasks.
* **Requires burst time estimation**: Like SJF and SRTF, estimating burst times is difficult and error-prone.

Characteristics:

* **Preemptive**: Like SRTF, processes with longer remaining burst times are executed first.
* **Starvation of short tasks**: Short tasks may be delayed indefinitely in favor of long tasks.

Trade-offs:

* **Fairness**: Unfair to short tasks, which may experience starvation.
* **Efficiency**: Potentially efficient in batch processing systems but inefficient for mixed workloads.
* **Real-time Responsiveness**: Poor for real-time tasks, as they could be delayed by long tasks.

**6. Multicore Highest Response Ratio Next (HRRN)**

Explanation:

**HRRN** is a non-preemptive algorithm that selects the process with the highest response ratio, which is calculated as:

Response Ratio=Waiting Time + Burst Time/Burst Time

This algorithm balances the need to prioritize short tasks while avoiding starvation for longer tasks. In multicore systems, HRRN can be applied to each core individually, with processes assigned based on their response ratios.

Advantages:

* **Balances waiting and burst times**: Minimizes waiting time while preventing starvation of long tasks.
* **Fair**: Ensures all processes get a fair chance of execution.
* **Adaptive**: Adapts to workload variations by dynamically adjusting process priorities.

Disadvantages:

* **Overhead**: Requires calculating response ratios continuously, which can introduce overhead.
* **Less optimal for real-time systems**: Can delay time-critical tasks if higher response ratio tasks are given priority.

Characteristics:

* **Non-preemptive**: Once a process starts, it runs to completion.
* **Fairness**: Balances fairness by dynamically adjusting priorities based on waiting time and burst time.
* **Adaptable**: Suitable for interactive and mixed workloads.

Trade-offs:

* **Fairness**: More fair than FCFS, SJF, and SRTF as it avoids starvation.
* **Efficiency**: Efficient for systems with mixed workloads but introduces overhead from response ratio calculations.
* **Real-time Responsiveness**: Less ideal for real-time systems, as processes with high response ratios could delay more time-sensitive tasks.

**7. Multicore Multilevel Queue Scheduling**

Explanation:

In **Multilevel Queue Scheduling (MLQ)**, processes are divided into different priority queues based on some attributes (such as **interactive** vs **CPU-bound**). In a **multicore system**, each core can handle processes from one or more queues, with higher-priority queues getting served first. Processes in the **interactive queue** might be scheduled on specific cores optimized for responsiveness, while **CPU-bound** processes are scheduled on other cores to utilize resources efficiently.

Advantages:

* **Prioritization**: Prioritizes certain types of processes, ensuring responsive handling of interactive tasks.
* **Improved system throughput**: By distributing processes into queues based on their characteristics, it optimizes resource utilization.

Disadvantages:

* **Starvation**: Processes in lower-priority queues may suffer from starvation if higher-priority queues keep filling up.
* **Complexity**: Managing multiple queues and adjusting priorities across cores adds complexity to the scheduler.
* **Hard to define boundaries**: Categorizing tasks accurately into queues can be difficult in real-time.

Characteristics:

* **Non-preemptive or preemptive**: Depends on the specific implementation for different queues.
* **Priority-based**: Processes are assigned to queues based on their characteristics.
* **Multiple queues**: Each queue may have a different scheduling algorithm, such as **Round Robin** for interactive tasks and **FCFS** for CPU-bound tasks.

Trade-offs:

* **Fairness**: May lead to unfairness for tasks in lower-priority queues due to starvation.
* **Efficiency**: High efficiency for mixed workloads (interactive and CPU-bound tasks).
* **Real-time Responsiveness**: Provides good real-time responsiveness for interactive tasks but may cause delays for lower-priority tasks.

**8. Multicore Multilevel Feedback Queue (MLFQ)**

Explanation:

**Multilevel Feedback Queue (MLFQ)** is an enhancement of Multilevel Queue Scheduling where processes can dynamically move between queues based on their behavior (e.g., CPU-bound vs I/O-bound). This makes the algorithm more adaptive to workload changes. In a multicore system, MLFQ can allocate processes across cores dynamically, moving tasks between queues depending on their execution characteristics.

Advantages:

* **Adaptable**: Adjusts to workloads dynamically, ensuring that CPU-bound tasks are given more CPU time and I/O-bound tasks are given less.
* **Improves fairness**: Avoids starvation by allowing processes to move between queues.
* **Efficient for varied workloads**: Balances the system's performance by optimizing for both CPU-bound and I/O-bound tasks.

Disadvantages:

* **Complexity**: Managing dynamic movement between queues can increase complexity.
* **Context-switching overhead**: The process of moving tasks between queues and rescheduling them can add overhead.
* **Hard to tune**: The parameters, like time quantum and the number of queues, must be tuned correctly to avoid inefficiencies.

Characteristics:

* **Preemptive**: Tasks can be preempted and moved between queues.
* **Adaptive**: Processes move between queues based on their observed behavior.
* **Multiple queues**: Allows fine-tuned control by having different queues for tasks with different characteristics.

Trade-offs:

* **Fairness**: Fairer than static multilevel queue systems since processes can move between queues.
* **Efficiency**: High efficiency for both interactive and CPU-bound tasks, but can incur overhead due to context switching.
* **Real-time Responsiveness**: Provides good responsiveness, especially for interactive tasks, but may introduce delays in lower-priority queues.

**9. Work-Stealing Scheduling**

Explanation:

**Work-Stealing** is a dynamic scheduling strategy used in multicore systems where idle cores "steal" work from other busy cores. When a core becomes idle, it picks up a task from another core’s queue (steals work) to ensure that all cores are fully utilized.

Advantages:

* **Load balancing**: Helps ensure that cores are equally loaded, preventing any core from being underutilized.
* **Adaptive**: Work-stealing is adaptive to the workload, as cores can steal work dynamically.

Disadvantages:

* **Overhead**: Work-stealing can introduce overhead due to the frequent stealing of tasks between cores.
* **Latency**: There may be some latency in work-stealing, especially when the tasks are large and require complex coordination.

Characteristics:

* **Dynamic**: Cores adapt based on the system’s workload.
* **Preemptive**: Tasks can be preempted and reassigned to different cores.
* **Efficient for parallel tasks**: Works well when the system has parallelizable workloads, like in multithreading scenarios.

Trade-offs:

* **Fairness**: Fairness is achieved by distributing work across all available cores, but high-priority tasks may sometimes be delayed if they are far from the idle core.
* **Efficiency**: Improves efficiency by utilizing idle cores but can introduce overhead from the stealing mechanism.
* **Real-time Responsiveness**: Real-time responsiveness may be impacted because work-stealing could introduce delays in task execution.

**10. Hybrid Scheduling Algorithms**

Explanation:

A **hybrid scheduling algorithm** combines multiple scheduling techniques to exploit the benefits of each. For example, a hybrid system might use Round Robin for interactive tasks and Shortest Job First (SJF) for CPU-bound tasks. In multicore systems, the hybrid approach could involve dynamically allocating tasks to different cores based on workload characteristics.

Advantages:

* **Flexibility**: Offers a flexible solution by combining different strategies.
* **Better performance for mixed workloads**: Works well in systems with varied workloads, providing the best of both worlds (i.e., good performance for interactive and CPU-bound tasks).

Disadvantages:

* **Complexity**: Managing different scheduling strategies for different tasks can be complex.
* **Tuning required**: The hybrid system may need frequent tuning to achieve optimal performance.

Characteristics:

* **Adaptive**: Combines multiple algorithms depending on the workload type.
* **Multiple techniques**: Often involves using different algorithms for different types of processes or tasks.
* **Multicore load balancing**: Tasks are dynamically distributed to different cores based on their characteristics.

Trade-offs:

* **Fairness**: Fairness depends on the algorithm used for different queues. Starvation can still occur if the priority system is poorly designed.
* **Efficiency**: Provides better efficiency than using a single algorithm across the board, as it tailors the scheduling to specific needs.
* **Real-time Responsiveness**: Can offer good real-time responsiveness by prioritizing interactive tasks, but care must be taken to avoid delays in non-prioritized tasks.

**11. Earliest Deadline First (EDF) for Multicore Systems**

Explanation:

**Earliest Deadline First (EDF)** is a real-time scheduling algorithm where processes are prioritized based on their deadlines. In a multicore system, EDF can be applied by assigning processes to cores based on which task has the earliest deadline. The task with the earliest deadline is given the CPU next.

Advantages:

* **Real-time efficiency**: Guarantees that real-time tasks meet their deadlines.
* **Optimal for deadline-based tasks**: In systems where tasks have strict deadlines, **EDF** ensures that tasks are scheduled in order of urgency.

Disadvantages:

* **Complexity**: Requires managing deadlines and task priorities across multiple cores.
* **Context switching overhead**: Due to the frequent priority adjustments, context switching overhead can be high.
* **Starvation**: Non-real-time tasks can be starved if real-time tasks keep arriving with deadlines.

Characteristics:

* **Preemptive**: Tasks can be preempted based on their deadlines.
* **Real-time scheduling**: Designed for systems where tasks have specific deadlines.
* **Fairness**: The focus is on meeting deadlines, so fairness for non-time-critical tasks may be compromised.

Trade-offs:

* **Fairness**: Less fair to non-real-time tasks, which may be delayed.
* **Efficiency**: Efficient in real-time systems but may result in high overhead in non-real-time systems.
* **Real-time Responsiveness**: Excellent real-time responsiveness, as tasks are always scheduled based on their deadlines.

**12. Priority-based Scheduling (Static and Dynamic)**

Explanation:

In **priority-based scheduling**, tasks are assigned a priority, and the process with the highest priority is scheduled first. This can be either static (priority remains fixed) or dynamic (priority changes based on process behavior or aging). In multicore systems, tasks can be distributed to cores based on their priority level.

Advantages:

* **Real-time prioritization**: Guarantees that high-priority tasks are executed first.
* **Flexibility**: Dynamic priority schemes adjust according to task behavior, which can improve system performance.

Disadvantages:

* **Starvation**: Low-priority tasks may starve if higher-priority tasks keep arriving.
* **Overhead**: Managing priorities, especially in dynamic systems, adds complexity and overhead.

Characteristics:

* **Preemptive or non-preemptive**: Can be configured to preempt lower-priority tasks or not.
* **Prioritized execution**: Tasks are executed based on their priority level.
* **Dynamic or static**: Priorities can either be fixed or adjusted during execution.

Trade-offs:

* **Fairness**: Can lead to unfairness due to starvation of lower-priority tasks.
* **Efficiency**: Highly efficient for real-time and prioritized workloads but may cause inefficiency in mixed workloads.
* **Real-time Responsiveness**: Excellent for real-time systems but can compromise efficiency for non-prioritized tasks.

**13. Lottery Scheduling**

Explanation:

**Lottery Scheduling** is a probabilistic scheduling algorithm where processes are assigned "lottery tickets," and the process that wins the lottery is executed next. In multicore systems, processes are assigned to cores based on the lottery outcome.

Advantages:

* **Fair**: Provides probabilistic fairness in terms of resource allocation.
* **Simple to implement**: Easy to implement and understand, even for large numbers of processes.

Disadvantages:

* **Unpredictable**: Since the lottery is random, the execution order can be unpredictable.
* **Overhead**: The need to manage and draw lottery tickets introduces overhead.
* **Not ideal for real-time systems**: The randomness may delay time-critical tasks.

Characteristics:

* **Probabilistic fairness**: Fairness is determined by the number of lottery tickets a process holds.
* **Non-preemptive**: Once a process starts executing, it runs to completion, though new tickets are drawn continuously.
* **Simple**: The algorithm is simple to implement and can handle large numbers of processes.

Trade-offs:

* **Fairness**: Provides fair resource allocation, but processes can still starve if they don’t win the lottery often.
* **Efficiency**: Can be inefficient for time-sensitive workloads due to randomness.
* **Real-time Responsiveness**: Poor for real-time tasks because of the unpredictability in task selection.

**14.Clustered Scheduling**

Explanation:

**Clustered Scheduling** is a type of scheduling strategy where the system is divided into clusters of processors or cores. Each cluster operates semi-independently with its own scheduler, which schedules tasks locally within the cluster. These clusters may be formed based on various factors like geographical proximity, resource affinity, or communication costs. A master scheduler may exist to allocate tasks between clusters, but the scheduling within each cluster remains isolated.

In multicore systems, this concept often extends to NUMA (Non-Uniform Memory Access) systems, where processors have local memory (which they can access faster than remote memory). Clusters are formed based on local memory groups, and each core within the cluster schedules tasks that are better suited to that core's memory.

Advantages:

* **Reduced overhead**: By limiting task scheduling to clusters, this approach reduces the need for complex global synchronization, which can be costly in terms of performance.
* **Improved locality**: Tasks can be assigned to processors within the same cluster, ensuring better data locality and cache efficiency. This reduces the time spent accessing remote memory, which is particularly useful in NUMA systems.
* **Scalability**: Scales better than global scheduling in large systems because each cluster operates semi-independently, avoiding the bottleneck of a single global scheduler.

Disadvantages:

* **Load balancing issues**: Clusters may become imbalanced if certain clusters are under-utilized while others are overburdened, leading to inefficiency.
* **Inter-cluster communication**: Communication between tasks assigned to different clusters may introduce latency and slow down system performance.
* **Less flexibility**: Since each cluster has its own scheduler, it may be less adaptable to workloads that require resources from multiple clusters.

Characteristics:

* **Local scheduling**: Each cluster has its own scheduler that schedules tasks based on local resources.
* **Communication overhead**: There is less inter-cluster communication, but it can still be an issue if tasks need to interact frequently with other clusters.
* **Autonomy of clusters**: Each cluster operates independently, which makes them relatively self-contained but can lead to underutilization or inefficiency if resources are not balanced.

Trade-offs:

* **Fairness**: Fairness within each cluster is ensured, but fairness between clusters may be harder to maintain. A poorly balanced workload across clusters can result in some cores being idle while others are overloaded.
* **Efficiency**: More efficient in terms of local task execution due to better data locality, but it may reduce overall system efficiency if the load is not well distributed.
* **Real-time Responsiveness**: Can be good for systems where tasks benefit from **local memory access** and low communication latency, but the scheduling delay across clusters could hinder real-time tasks requiring frequent inter-cluster communication.

**15.Global Scheduling**

Explanation:

In **Global Scheduling**, the system has a single global scheduler that manages task allocation across all available cores or processors in the system. All tasks, regardless of which processor they are running on, are treated as part of a global pool of processes, and decisions about which core will execute a task are made by this central scheduler. This strategy is typical in homogeneous multicore systems where all cores have equal access to shared memory, and task distribution can be done dynamically.

Global scheduling is often associated with systems that do not have the complexities of NUMA architecture or systems with symmetric multiprocessors (SMP). Preemptive global scheduling ensures tasks are moved across processors to maximize resource utilization, minimize waiting times, and balance loads effectively.

Advantages:

* **Better load balancing**: A global scheduler can dynamically redistribute tasks to prevent cores from becoming underutilized while others are overloaded, leading to better resource utilization and system efficiency.
* **Higher flexibility**: It allows the scheduler to move tasks freely across cores, providing a high degree of flexibility, which is useful in heterogeneous environments.
* **Simplified management**: With a global scheduler, there is no need to coordinate between different local schedulers or clusters, which simplifies the overall system design.

Disadvantages:

* **Global contention**: A single global scheduler can become a bottleneck, especially in large systems with many cores, leading to contention for the scheduler’s decision-making process.
* **Higher communication overhead**: If tasks need to be moved frequently across cores, it can lead to communication overhead and latency, especially in systems with NUMA architectures.
* **Context-switching overhead**: Frequent switching of tasks between cores can lead to increased **context switching** overhead, reducing system efficiency.

Characteristics:

* **Single scheduler**: One central scheduler manages all task allocation across all cores.
* **Preemptive**: The system is often preemptive, allowing tasks to be shifted between cores as needed.
* **Load balancing**: The global scheduler can move tasks to underutilized cores, ensuring balanced resource utilization.

Trade-offs:

* **Fairness**: The global scheduling algorithm can ensure fairness by considering all tasks in the system equally, but it may introduce starvation issues for certain tasks if load balancing is not optimal.
* **Efficiency**: Global scheduling can achieve high efficiency in systems with homogeneous processors and workloads that are evenly distributed. However, the scheduler overhead may reduce efficiency in large systems or systems with complex workloads.
* **Real-time Responsiveness**: Global scheduling might hinder real-time responsiveness due to the time taken to decide which tasks should be moved across cores, especially in high-performance systems with strict timing requirements.

**Result:**

Below is the comparative table for the given CPU scheduling algorithms in multicore systems, focusing on fairness, real-time responsiveness, efficiency, and overall performance. These evaluations depend on general characteristics and trade-offs of the algorithms.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Fairness** | **Real-time**  **Responsiveness** | **Efficiency** | **Overall**  **Performance** | **Description** |
| 1. **Multicore Round Robin (RR)** | high | low | Medium | Medium | Round Robin allocates time slices to processes in a circular order, ensuring fairness but low responsiveness due to fixed time slices. |
| 1. **Multicore FCFS** | low | low | Medium | Low | FCFS processes in arrival order, which may cause long wait times and poor responsiveness, with low efficiency in handling various workloads. |
| 1. **Multicore SJF** | Medium | Low | High | Medium | Shortest Job First minimizes waiting time by executing shortest tasks first but is non-preemptive and less responsive for real-time tasks. |
| 1. **Multicore SRTF** | Medium | Medium | High | High | Shortest Remaining Time First preempts tasks to minimize remaining execution time, improving efficiency but potentially lowering responsiveness. |
| 1. **Multicore LRTF** | Low | Low | Medium | Low | Longest Remaining Time First favors long tasks, which may lead to poor fairness and responsiveness. |
| 1. **Multicore HRRN** | Medium | Low | High | Medium | Highest Response Ratio Next prioritizes tasks based on their response ratio, offering better fairness and efficiency but not real-time responsiveness. |
| 1. **Multicore Multilevel Queue** | Medium | Medium | Medium | Medium | Tasks are divided into multiple queues, with each queue having its own scheduling algorithm; it improves organization but can suffer from fairness issues. |
| 1. **Multicore MLFQ** | High | Medium | High | High | Multilevel Feedback Queue adapts to the behavior of tasks, providing fairness and efficiency but moderate responsiveness for real-time tasks. |
| 1. **Work-Stealing Scheduling** | High | Medium | High | High | Tasks are dynamically distributed among cores to balance load, improving efficiency and fairness but with moderate real-time responsiveness. |
| 1. **Hybrid Scheduling Algorithms** | High | High | High | High | Combines multiple scheduling techniques to optimize fairness, efficiency, and responsiveness, suitable for diverse workloads. |
| 1. **EDF for Multicore Systems** | Low | High | Medium | High | Earliest Deadline First schedules tasks based on deadlines, making it ideal for real-time systems but less efficient |
| 1. **Priority-based Scheduling** | Low | High (Dynamic) / Low (Static) | Medium | Medium | Prioritizes tasks based on predefined or dynamic priorities, ensuring fast response for high-priority tasks but lower fairness in static systems. |
| 1. **Lottery Scheduling** | High | Low | Medium | Medium | Lottery Scheduling assigns probabilities to tasks for selection, ensuring fairness but lower responsiveness due to randomness. |
| 1. **Clustered Scheduling** | Medium | Medium | High | Medium | Tasks are grouped into clusters, each using a separate scheduler, improving efficiency but can lead to fairness issues between clusters. |
| 1. **Global Scheduling** | High | Medium | High | High | Global scheduling schedules all tasks across cores, offering load balancing and fairness, but may suffer from medium real-time responsiveness. |

**Conclusion:**

In conclusion, this research underscores the complexity of designing CPU scheduling algorithms for multicore systems that can simultaneously address the demands for real-time responsiveness, efficiency, and fairness. While existing algorithms excel in certain areas, they often fall short when it comes to balancing all three factors, especially in the dynamic and diverse environments of modern multicore systems The research suggests that future work should focus on adaptive and hybrid solutions, as well as improving the ability of systems to learn from past behavior to optimize performance. As the demands for performance, responsiveness, and fairness continue to grow, developing sophisticated algorithms capable of handling these trade-offs will be critical for optimizing both real-time and general-purpose computing systems in the future. These solutions are expected to be crucial as the demands for higher performance, betterresponsiveness, and fair resource allocation continue to rise in both real-time and general-purpose computing systems.

**REFERNECES:**

<https://www.sciencedirect.com/topics/computer-science/multicore-system#:~:text=A%20multicore%20system%20refers%20to,multiple%20boards%20in%20a%20system>.

<https://www.geeksforgeeks.org/difference-between-multicore-and-multiprocessor-system/>

<https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/>

<https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html>

<https://www.javatpoint.com/cpu-scheduling-algorithms-in-operating-systems>

<https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm>

<https://en.wikipedia.org/wiki/Scheduling_(computing)>

<https://www.javatpoint.com/os-scheduling-algorithms>

<https://www.linkedin.com/advice/0/what-different-types-scheduling-algorithms-used-teabe>

<https://aissmspoly.org.in/wp-content/uploads/2020/01/CPU-Scheduling-and-Algorithm-.pdf>

<https://www.tutorialspoint.com/operating_system/pdf/os_process_scheduling_algorithms.pdf>

<https://www.researchgate.net/publication/356513035_Scheduling_and_Synchronization_Algorithms_in_Operating_System_A_Survey>

<https://en.wikipedia.org/wiki/Round-robin_scheduling>

<https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/>

<https://unstop.com/blog/scheduling-algorithms-in-operating-system>

<https://byjus.com/gate/process-scheduling-in-operating-system-notes/>

<https://www.javatpoint.com/os-process-schedulers>

**Books references:**

* Operating Systems , William Stallings, Latest Edition
* Operating System Principles ,Abraham Silberschatz, Peter Baer Galvin, Greg Gagne

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***