Nepal College of Information Technology

Affiliated to Pokhara University

Advanced Process Scheduling Techniques

Rishav Raj
ME Computer

Introduction to Process Scheduling

Process scheduling is the process of selecting and executing processes in an operating system. It involves managing the order and timing of processes, and which processes are allowed to run.

Importance of process scheduling

1. Efficient CPU Utilization
2. Improved System Performance
3. Fairness and Prioritization
4. Enhanced Responsiveness
5. Supports Multitasking and Multiprogramming
6. Energy Efficiency
7. Prevents Deadlocks and Starvation
8. Scalability and Adaptability

Basics of Traditional Scheduling Algorithms

First-Come, First-Served (FCFS)

- **Description:** Processes are executed in the order they arrive in the ready queue.
- **Advantages:** Simple and easy to implement.
- **Disadvantages:** Can lead to long waiting times, especially if a long process arrives first (known as the **convoy effect**).

Shortest Job Next (SJN) (also known as Shortest Job First - SJF)

- **Description:** The process with the shortest estimated execution time is selected next.
- **Advantages:** Minimizes average waiting time.
- **Disadvantages:** Requires knowledge of process execution time in advance and may cause starvation for longer processes.

Basics of Traditional Scheduling Algorithms

Round Robin (RR)

- **Description:** Each process is assigned a fixed time slice (quantum) and executed in cyclic order. If a process isn't finished in its time slot, it moves to the back of the queue.
- **Advantages:** Provides fairness and is well-suited for time-sharing systems.
- **Disadvantages:** Performance depends on the time quantum size; too small leads to excessive context switching, too large resembles FCFS.

Priority Scheduling

- **Description:** Processes are executed based on their priority level. Higher-priority processes run first.
- **Advantages:** Important tasks are handled promptly.
- **Disadvantages:** Lower-priority processes may suffer from **starvation** if higher-priority processes keep arriving. **Aging** techniques can be used to gradually increase the priority of waiting processes to avoid this.

Limitations of Traditional Scheduling Techniques

Starvation in Priority Scheduling

- **Issue:** In **Priority Scheduling**, low-priority processes may never get executed if higher-priority processes keep arriving. This indefinite postponement is known as **starvation**.
- **Solution: Aging** can be used to gradually increase the priority of waiting processes, ensuring they eventually get CPU time.

Inefficiency in FCFS for Short Tasks

- **Issue:** In **First-Come, First-Served (FCFS)**, long processes at the front of the queue can delay shorter tasks that arrive later. This leads to **high average waiting time** and the **convoy effect**, where shorter tasks are inefficiently delayed.
- **Impact:** Poor responsiveness in systems with mixed short and long tasks.

Limitations of Traditional Scheduling Techniques

Fixed Time Quantum Issues in Round Robin (RR)

- **Issue:** The performance of **Round Robin** heavily depends on the size of the **time quantum**:
    - **Too Small:** Causes excessive **context switching**, increasing overhead and reducing efficiency.
    - **Too Large:** Starts to behave like **FCFS**, increasing waiting time for short tasks.
- **Impact:** Finding the optimal quantum size is challenging, and a poor choice can degrade system performance.

Introduction to Advanced Scheduling Techniques

Advanced Scheduling Techniques are designed to overcome the limitations of traditional algorithms by improving efficiency, fairness, and responsiveness in modern computing environments. They adapt to dynamic workloads, prioritize real-time tasks, and ensure better resource management.

Need for Advanced Techniques

- **Address Limitations:** Traditional methods struggle with issues like **starvation**, **inefficient CPU utilization**, and **poor responsiveness** in complex systems.
- **Dynamic Workloads:** Modern systems handle diverse tasks, including **real-time processes**, requiring more adaptable scheduling strategies.
- **Resource Optimization:** Advanced techniques optimize CPU, memory, and power usage, critical for high-performance and mobile systems.

Focus on Improving Efficiency and Fairness

- **Efficiency:** Minimize **waiting time**, **turnaround time**, and **context switching** to ensure faster task execution and better resource utilization.
- **Fairness:** Ensure all processes get a fair share of CPU time, preventing issues like **starvation** while balancing **priority** and **performance** needs.

Multicore

Multicore processors require advanced scheduling techniques to efficiently distribute tasks across multiple cores, maximizing parallelism and system performance. Traditional single-core scheduling methods are not sufficient due to increased complexity in task allocation, load balancing, and inter-core communication.

Key Multicore Scheduling Techniques

1. **Symmetric Multiprocessing (SMP) Scheduling**
2. **Asymmetric Multiprocessing (AMP) Scheduling**
3. **Load Balancing Scheduling**
4. **Cache-Aware Scheduling**
5. **Energy-Aware Scheduling**
6. **Real-Time Multicore Scheduling**

Key Multicore Scheduling Techniques

1. **Symmetric Multiprocessing (SMP) Scheduling**

   Description: All cores share a common ready queue, and the OS assigns processes dynamically.
   Advantages: Simple and balanced workload distribution.
   Challenges: Potential contention when multiple cores access the shared queue.

2. **Asymmetric Multiprocessing (AMP) Scheduling**

   Description: A single core (master) handles scheduling and assigns tasks to worker cores.
   Advantages: Reduces scheduling overhead on worker cores.
   Challenges: The master core can become a bottleneck.

3. **Load Balancing Scheduling**

   Description: The OS dynamically redistributes tasks among cores to ensure equal workload distribution.
   Types:
   - **Push Migration:** Scheduler proactively moves tasks from overloaded to underloaded cores.
   - **Pull Migration:** Idle cores pull tasks from busy cores.

   Advantages: Prevents core overloading and optimizes performance.

Key Multicore Scheduling Techniques

4. Cache-Aware Scheduling

Description: Processes are scheduled on the same core they previously executed on to improve cache efficiency.
Advantages: Reduces cache misses and enhances performance.
Challenges: May cause load imbalance if some cores remain underutilized.

5. Energy-Aware Scheduling

Description: Adjusts task allocation and CPU frequency to optimize power consumption.
Techniques:

- **Dynamic Voltage and Frequency Scaling (DVFS)** reduces power use for less demanding tasks.
- **Big.LITTLE architecture** schedules tasks to high-performance or energy-efficient cores based on demand.

Advantages: Extends battery life in mobile devices and reduces energy costs in data centers.

6. Real-Time Multicore Scheduling

Description: Used in time-sensitive applications (e.g., embedded systems, automotive, robotics).
Common Methods:

- **Partitioned Scheduling:** Each core gets a fixed set of tasks.
- **Global Scheduling:** Tasks are scheduled dynamically across all cores.

Advantages: Ensures deadlines are met in real-time systems.

Affinity-Based Multicore Scheduling

Affinity-based scheduling is a technique used in **multicore processors** to improve performance by keeping processes bound to specific cores, reducing context-switching overhead and improving cache efficiency.

Types of Affinity-Based Scheduling

1. **Soft Affinity (Advisory Affinity)**
   - The scheduler prefers to keep a process on the same core but **can** migrate it if needed.
   - Balances performance and load distribution.
   - Used in **general-purpose OS scheduling** like Linux CFS (Completely Fair Scheduler).
2. **Hard Affinity (Strict Affinity)**
   - A process is strictly bound to a specific core and **cannot** be moved.
   - Reduces inter-core communication but may lead to **load imbalance**.
   - Used in **real-time and specialized systems** like embedded systems and HPC (High-Performance Computing).

Affinity-Based Multicore Scheduling

Advantages of Affinity-Based Scheduling

Cache Efficiency: Reduces cache misses by keeping processes on the same core.
Lower Context-Switch Overhead: Avoids unnecessary task migrations between cores.
Better Performance for Multithreading: Enhances execution for CPU-intensive applications.
Energy Efficiency: Reduces power consumption by limiting inter-core communication.

Challenges

Load Imbalance: Some cores may be overburdened while others remain idle.
Reduced Flexibility: Hard affinity can cause inefficiencies in dynamic workloads.
Complex Implementation: Managing affinity dynamically requires intelligent scheduling policies.

Use Cases

- **Operating Systems (Linux, Windows):** Implements soft affinity to optimize performance.
- **High-Performance Computing (HPC):** Uses hard affinity for scientific and AI workloads.
- **Real-Time Systems:** Ensures predictable execution times for critical tasks.

Load Balancing in Multicore Scheduling

Load balancing in multicore scheduling ensures that tasks are evenly distributed across multiple CPU cores, preventing some cores from being overloaded while others remain idle. This improves system efficiency, reduces task waiting times, and enhances overall performance.

Types of Load Balancing Techniques

1. **Static Load Balancing**
   - Tasks are assigned to cores at the **start of execution** and do not migrate.
   - **Example:** Partitioned scheduling in real-time systems.
   - **Advantage:** Low overhead, predictable performance.
   - **Disadvantage:** May cause **imbalance** if some cores finish tasks earlier than others.
2. **Dynamic Load Balancing**
   - The scheduler **actively moves tasks** between cores based on workload changes.
   - **Example:** Linux CFS (Completely Fair Scheduler), Windows scheduler.
   - **Advantage:** Adapts to system workload, avoids idle cores.
   - **Disadvantage:** Higher **overhead** due to frequent migrations.

Load Balancing in Multicore Scheduling

Load Balancing Strategies

1. **Push Migration**
   - An **overloaded core pushes** tasks to an underloaded core.
   - **Advantage:** Proactive balancing prevents bottlenecks.
   - **Example:** Used in Linux kernel's load balancer.
2. **Pull Migration**
   - An **idle core pulls** tasks from a busy core.
   - **Advantage:** Efficient use of available processing power.
   - **Example:** Used in Windows thread scheduler.
3. **Work Stealing**
   - Each core maintains its own task queue; an idle core **steals** tasks from busy cores.
   - **Advantage:** Scalable for parallel applications.
   - **Example:** Used in Java's Fork/Join framework.

Load Balancing in Multicore Scheduling

Challenges in Multicore Load Balancing

Overhead: Task migration adds scheduling overhead.
Cache Performance Loss: Moving tasks between cores may lead to **cache misses**.
Power Consumption: Frequent migrations increase **energy usage** in mobile and embedded systems.

Use Cases

General-Purpose OS (Linux, Windows): Balances CPU load for smooth multitasking.
Cloud Computing & Data Centers: Optimizes resource allocation in virtualized environments.
High-Performance Computing (HPC): Distributes large-scale parallel workloads efficiently.

Cache and Memory Optimization in Multicore Scheduling

Efficient scheduling in multicore systems must consider **cache locality** and **memory access patterns** to minimize cache misses, reduce memory latency, and improve overall performance.

Key Techniques for Cache and Memory Optimization in Scheduling

Cache-Aware Scheduling

- **Goal:** Minimize cache misses by keeping processes on the same core where their data is cached.
- **Technique:** Assigns processes to cores **based on their last execution** (CPU affinity).
- **Example: Linux Completely Fair Scheduler (CFS)** uses CPU affinity to optimize cache reuse.

Advantage: Improves execution speed by reducing cache warm-up time.

Challenge: Can lead to load imbalance if some cores become overloaded.

Cache and Memory Optimization in Multicore Scheduling

NUMA-Aware Scheduling (Non-Uniform Memory Access)

- **Goal:** Optimize memory access in NUMA architectures, where different cores have **different memory access speeds**.
- **Technique:** Ensures a process runs on the same NUMA node (set of cores with shared memory) to **reduce remote memory access latency**.
- **Example: Linux NUMA Balancing** dynamically migrates processes closer to their data.

Advantage: Reduces memory access time, improving performance in **high-memory workloads**.
Challenge: Requires dynamic monitoring and migration, increasing scheduling complexity.

Memory Bandwidth-Aware Scheduling

- **Goal:** Prevent memory bottlenecks by distributing memory-intensive tasks across cores.
- **Technique:** Identifies memory-heavy processes and spreads them to **avoid contention** on memory channels.
- **Example:** Used in **cloud computing** and **HPC schedulers** like SLURM.

Advantage: Reduces contention for memory bandwidth, improving system responsiveness.
Challenge: Difficult to predict memory usage dynamically.

Cache and Memory Optimization in Multicore Scheduling

Prefetching-Aware Scheduling

- **Goal:** Improve CPU performance by scheduling tasks based on hardware **prefetching mechanisms**.
- **Technique:** Schedulers prioritize tasks with predictable memory access patterns to **align with hardware prefetching**.
- **Example:** Used in AI workloads for deep learning training.

Advantage: Reduces cache miss rates for **data-intensive applications**.
Challenge: Prefetching behavior varies across different processors, requiring custom tuning.

Cache and Memory Optimization in Multicore Scheduling

Challenges in Cache and Memory-Aware Scheduling

Balancing Cache Affinity and Load Distribution – Keeping tasks on the same core (for cache benefits) may cause imbalance.
Memory Latency Variations in NUMA Systems – Some cores may experience longer memory access times than others.
Overhead of Dynamic Migration – Continuously moving tasks to optimize memory use **increases scheduler complexity**.

Use Cases

High-Performance Computing (HPC): Optimizes large-scale simulations and deep learning workloads.
Cloud Computing: Ensures efficient memory and cache usage in virtualized environments.
Real-Time Systems: Reduces memory access delays in automotive and robotics applications.

NUMA-Aware Scheduling (Non-Uniform Memory Access)

Non-Uniform Memory Access (NUMA) is a memory architecture used in modern multicore and multiprocessor systems, where different processors have varying memory access speeds based on proximity to memory nodes. NUMA-aware scheduling ensures that processes execute on CPU cores closer to their allocated memory, reducing latency and improving performance.

Why is NUMA-Aware Scheduling Important?

- **Reduces Memory Access Latency** – Ensures tasks access local memory rather than slower remote memory.
- **Improves Cache Performance** – Keeps processes on the same CPU to maximize cache reuse.
- **Prevents Memory Bandwidth Bottlenecks** – Distributes memory-intensive tasks efficiently.
- **Enhances Scalability** – Essential for large-scale systems like **HPC, cloud computing, and database servers**.

NUMA-Aware Scheduling

How NUMA-Aware Scheduling Works

1️⃣**Process-to-Memory Affinity**

- A process is scheduled on a CPU **near its allocated memory region**.
- This avoids unnecessary **remote memory access**, which is slower.

2️⃣**Dynamic NUMA Balancing**

- The scheduler **monitors memory access patterns** and **migrates processes** closer to frequently accessed memory.
- **Example:** Linux Kernel's **AutoNUMA** dynamically moves tasks and memory pages.

3️⃣**Load Balancing Across NUMA Nodes**

- Ensures workload is evenly spread across NUMA nodes to prevent bottlenecks.
- **Example:** Windows Scheduler dynamically redistributes tasks between NUMA nodes.

4️⃣**Thread and Core Affinity Management**

- In multi-threaded applications, **threads are kept on the same NUMA node** to maximize shared memory efficiency.
- **Example:** Used in databases like **Oracle DB and SQL Server** for optimizing memory performance.

NUMA-Aware Scheduling (Non-Uniform Memory Access)

Challenges of NUMA-Aware Scheduling

Task Migration Overhead – Moving tasks between NUMA nodes can cause performance drops.
Balancing Load vs. Memory Locality – Ensuring **both** even CPU load **and** optimal memory access is complex.
Requires OS & Application Optimization – Applications must be NUMA-aware to fully benefit.

Use Cases of NUMA-Aware Scheduling

High-Performance Computing (HPC): Scientific simulations, AI workloads.
Database Servers: SQL Server, Oracle DB, PostgreSQL optimize queries using NUMA-awareness.
Cloud Computing & Virtualization: VMware ESXi, Kubernetes use NUMA-aware scheduling for VM placement.
Gaming & Graphics Processing: Optimizes CPU-intensive game engines.

Reference

https://www.google.com/search?q=definition+of+process+scheduling&oq=Definition+of+Process+Scheduling&gs_lcrp=EgZjaHJvbWUqBwgAEAAYgAQyBwgAEAAYgAQyCAgBEAAYFhgeMggIAhAAGBYYHjIICAMQABgWGB4yCAgEEAAYFhgeMg0IBRAAGIYDGIAEGIoFMg0IBhAAGIYDGIAEGIoFMg0IBxAAGIYDGIAEGIoFMg0ICBAAGIYDGIAEGIoFMgcICRAAGO8F0gEIMTAyOWowajeoAgCwAgA&sourceid=chrome&ie=UTF-8

https://www.cs.ucr.edu/~nael/cs202/lectures/lec7.pdf

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

https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm

https://www.mdpi.com/2076-3417/11/12/5740#:~:text=The%20affinity%2Dbased%20scheduling%20of,overall%20performance%20of%20the%20system

Thank You!