Chapter 10: Multiprocessor Scheduling (Advanced)

This chapter explores the challenges and strategies for scheduling processes across multiple CPUs. With the rise of multicore processors, understanding how to effectively manage multiple CPUs is critical. This topic builds on both single-CPU scheduling and concurrency principles.

> **The Crux of the Problem: How to Schedule Jobs on Multiple CPUs?**
>
> How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work, or are new ideas required?

1. Background: Multiprocessor Architecture

The key difference between single-CPU and multi-CPU systems lies in how **hardware caches** and **shared memory** interact.

1.1. Caches and Locality

- **Caches:** Small, fast memories that hold copies of frequently used data from the slower main memory. They improve performance by exploiting two types of locality:
  - **Temporal Locality:** Data that is accessed once is likely to be accessed again soon.
  - **Spatial Locality:** Data near a recently accessed address is likely to be accessed soon.

1.2. The Cache Coherence Problem

When multiple CPUs each have their own private cache but share a single main memory, a problem arises: how to ensure all CPUs have a consistent, up-to-date view of memory?

- **Scenario:**
  1. Process on CPU 1 reads address `A` (value `D`) into its cache.
  2. Process on CPU 1 updates the value in its cache to `D'`. This update might not be immediately written back to main memory.
  3. Process is moved to CPU 2 and reads address `A`. CPU 2's cache fetches the value from main memory, retrieving the old, stale value `D` instead of the correct value `D'`.
- **Hardware Solution:** Modern hardware solves this with **cache coherence protocols**. For example, using **bus snooping**, each cache monitors the main memory bus. When it sees an update to a memory location it has cached, it can either **invalidate** (discard) its old copy or **update** it with the

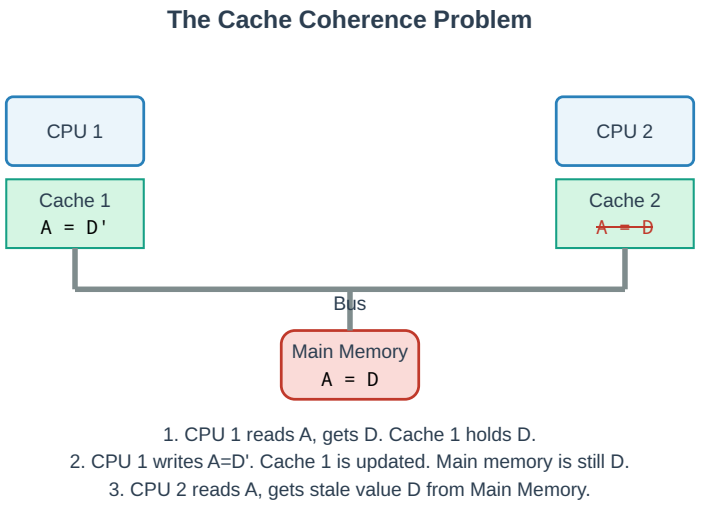


Figure 1: The Cache Coherence Problem

new value. This ensures that, from a programmer's perspective, memory appears as a single, consistent entity.

1.3. Synchronization

Even with hardware cache coherence, accessing shared data structures across multiple CPUs is not automatically safe.

- **The Need for Locks:** Coherence protocols ensure that a single memory load or store is consistent, but they do not protect multi-instruction operations. To correctly modify a shared data structure (like a linked list or queue), you must use synchronization primitives like **locks** (`mutexes`) to ensure operations are **atomic**.
- **Example:** Without a lock, two threads on different CPUs trying to remove an element from the head of a linked list at the same time could both read the same `head` pointer, leading to data corruption and errors.

1.4. Cache Affinity

- **Concept:** When a process runs on a CPU, it populates that CPU's caches with its data and instructions (it "warms up" the cache).
- **Performance Implication:** It is advantageous to keep a process running on the **same CPU** whenever possible. If a process is constantly moved between CPUs (migrated), it must repeatedly warm up the cache on the new CPU, leading to poor performance.

- **Scheduler Goal:** A multiprocessor scheduler should therefore try to respect **cache affinity**.

---

2. Multiprocessor Scheduling Approaches

2.1. Single-Queue Multiprocessor Scheduling (SQMS)

The simplest approach is to use a single, global queue for all jobs, shared among all CPUs.

- **Mechanism:** When a CPU becomes idle, it picks the next best job from the single queue to run.
- **Advantages:**
  - **Simplicity:** Easy to adapt from a single-CPU scheduler.
  - **Load Balancing:** Load is naturally balanced, as there is never an idle CPU if there are jobs in the queue.
- **Disadvantages:**
  - **Scalability:** The single queue is a shared resource and requires locking for safe access. As the number of CPUs increases, contention for this lock becomes a major performance bottleneck.
  - **Cache Affinity:** This approach is inherently poor for cache affinity. Since any idle CPU can pull any job, processes tend to "bounce" between CPUs, constantly invalidating their cache state.

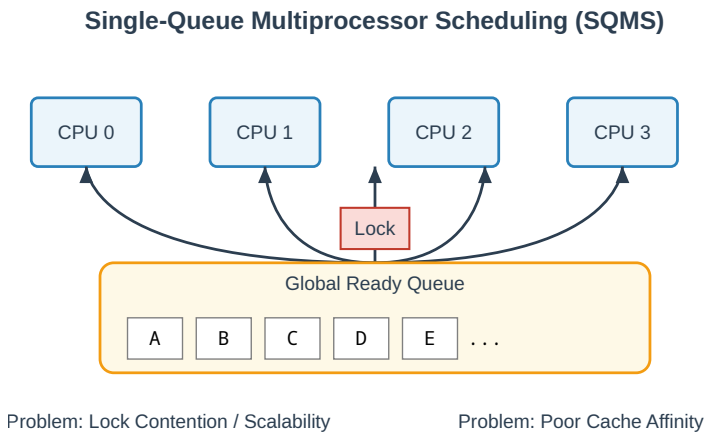


Figure 2: Single-Queue Multiprocessor Scheduling (SQMS)

2.2. Multi-Queue Multiprocessor Scheduling (MQMS)

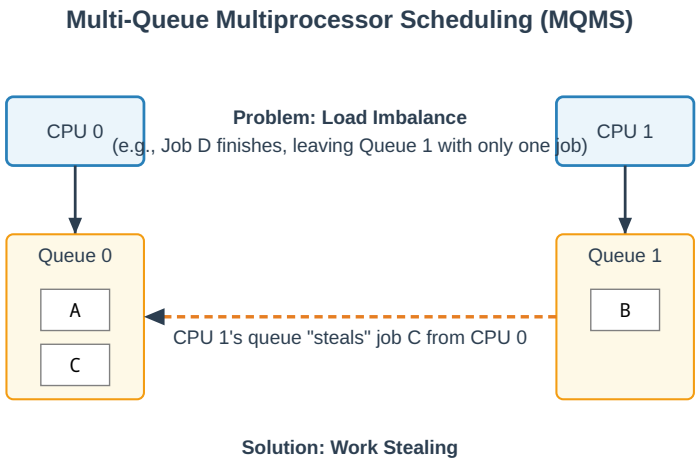
To overcome the limitations of SQMS, most modern systems use a multi-queue approach, typically with one queue per CPU.

- **Mechanism:** Each CPU has its own private queue of jobs. When a new job enters the system, it is assigned to a specific queue. Each CPU schedules independently from its own queue.
- **Advantages:**
  - **Scalability:** With no shared queue, the need for locking is greatly reduced, avoiding the single-lock bottleneck.
  - **Cache Affinity:** Jobs naturally stay on the same CPU, preserving cache state and improving performance.
- **The New Problem: Load Imbalance**
  - MQMS introduces a new, fundamental problem. If one CPU's queue becomes empty while another's is full, one CPU will sit idle while there is still work to be done. This leads to inefficient resource utilization.

  The Crux of the Problem: How to Deal with Load Imbalance?

  How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?

- **Solution: Migration**
  - To solve load imbalance, the OS must be able to **migrate** jobs from an overworked CPU's queue to an underworked (or idle) one.
  - **Work Stealing:** A common technique where an idle or under-loaded queue ("source") "steals" a job from an overloaded queue ("target").
  - **The Trade-off:** How often should a queue check others for work?
    * Too often: High overhead, negates the scalability benefits of MQMS.
    * Too infrequently: Risk of severe load imbalance.
    * Finding the right balance is a difficult policy decision.



3. Linux Multiprocessor Schedulers

The Linux kernel has seen several different multiprocessor schedulers over the years, demonstrating that there is no single "best" solution.

- **O(1) Scheduler:** A priority-based scheduler similar to MLFQ.
- **Completely Fair Scheduler (CFS):** A proportional-share scheduler similar to Stride Scheduling. (This is the current default).
- **BF Scheduler (BFS):** A single-queue, proportional-share scheduler based on a complex algorithm called EEVDF.

The existence of these different approaches highlights the complex trade-offs between scalability, load balancing, cache affinity, and the core scheduling goals (turnaround time, response time, fairness).

---

4. Summary

Multiprocessor scheduling introduces new challenges not present in single-CPU systems, primarily related to **cache coherence**, **synchronization**, and **cache affinity**.

- **Single-Queue (SQMS)** schedulers are simple and load-balance well but suffer from poor scalability and cache affinity.
- **Multi-Queue (MQMS)** schedulers scale well and preserve cache affinity but must implement a **migration** policy (like work stealing) to combat load imbalance.
- Building a general-purpose multiprocessor scheduler involves navigating these difficult trade-offs, and there is no one-size-fits-all solution.