# Advanced Process and Thread Management

Anil Verma

December 2024

## Lecture Notes: Advanced Process and Thread Management

## 1 Advanced Process Scheduling Techniques (2.1)

### 1.1 Multi-core Scheduling

Modern multi-core systems require advanced scheduling techniques to optimize performance. These techniques focus on minimizing context switching, reducing cache misses, and balancing the load across cores.

#### 1.1.1 Affinity-Based Scheduling

Affinity-based scheduling ensures that processes or threads are bound to specific CPU cores to improve performance. This technique is particularly useful in systems where cache locality and memory access patterns are critical.

- **CPU Affinity**: Binding a process to a specific CPU core reduces context switching overhead. For example, in Linux, the `sched_setaffinity` system call can be used to set CPU affinity. This is crucial in real-time systems where predictable performance is essential.
- **Cache Affinity**: Keeping a process on the same core leverages cached data, reducing cache misses. For example, if a process frequently accesses the same memory locations, it benefits from being scheduled on the same core. This is because the data remains in the cache, reducing the need to fetch it from main memory.
- **Memory Affinity**: Allocating memory close to the CPU core where the process runs minimizes memory access latency. This is especially important in NUMA (Non-Uniform Memory Access) architectures, where memory access times vary depending on the proximity of the memory to the CPU.

### 1.1.2 Numerical Example: Affinity-Based Scheduling

Consider a quad-core processor with 6 processes (P1, P2, P3, P4, P5, P6). Each process has a preferred core and a cache miss penalty if scheduled on a different core. The goal is to minimize cache misses and execution time.

- **Process Execution Times**: P1: 10 ms, P2: 15 ms, P3: 20 ms, P4: 25 ms, P5: 30 ms, P6: 35 ms.
- **Cache Affinity Requirements**: P1: Core 1, P2: Core 2, P3: Core 3, P4: Core 4, P5: Core 1, P6: Core 2.
- **Cache Miss Penalty**: 5 ms if scheduled on a non-preferred core.

### 1.1.3 Solution: Affinity-Based Scheduling

1. **Initial Assignment**: Assign each process to its preferred core if possible.
   - P1: Core 1
   - P2: Core 2
   - P3: Core 3
   - P4: Core 4
   - P5: Core 1 (conflict with P1)
   - P6: Core 2 (conflict with P2)
2. **Resolve Conflicts**:
   - P5 cannot be scheduled on Core 1 because P1 is already there. Schedule P5 on Core 3 (least penalty).
   - P6 cannot be scheduled on Core 2 because P2 is already there. Schedule P6 on Core 4 (least penalty).
3. **Calculate Execution Times with Penalties**:
   - P1: 10 ms (Core 1)
   - P2: 15 ms (Core 2)
   - P3: 20 ms (Core 3)
   - P4: 25 ms (Core 4)
   - P5: 30 ms + 5 ms penalty = 35 ms (Core 3)
   - P6: 35 ms + 5 ms penalty = 40 ms (Core 4)
4. **Total Execution Time**:
   - Core 1: 10 ms (P1)
   - Core 2: 15 ms (P2)
   - Core 3: 20 ms (P3) + 35 ms (P5) = 55 ms
   - Core 4: 25 ms (P4) + 40 ms (P6) = 65 ms
5. **Sum of Execution Times**: Total = 10 ms + 15 ms + 55 ms + 65 ms = 145 ms.

### 1.1.4 Load Balancing

Load balancing ensures that tasks are evenly distributed across CPU cores to prevent overloading and underutilization. There are two main types of load balancing:

- **Static Load Balancing**: Tasks are distributed based on a pre-determined algorithm. For example, Round-Robin scheduling assigns tasks to cores in a cyclic manner. This method is simple to implement but may not adapt well to dynamic workloads.
- **Dynamic Load Balancing**: Tasks are dynamically assigned to cores based on the current load. Techniques include:
  - **Least Loaded Scheduling**: Tasks are assigned to the core with the least current load. This method is more adaptive but requires real-time monitoring of core loads.
  - **Work Stealing**: Idle cores steal tasks from busy cores to balance the load. This method is effective in highly dynamic environments where workloads can vary significantly.

### 1.1.5 Numerical Example: Load Balancing

A system with 4 cores and 12 tasks is balanced using Round-Robin, Least Loaded Scheduling, and Work Stealing. The overall execution time is determined by the core with the longest execution time.

- **Task Execution Times**: T1: 10 ms, T2: 15 ms, T3: 20 ms, T4: 25 ms, T5: 30 ms, T6: 35 ms, T7: 40 ms, T8: 45 ms, T9: 50 ms, T10: 55 ms, T11: 60 ms, T12: 65 ms.
- **Static Load Balancing (Round-Robin)**: Core 1: T1, T5, T9; Core 2: T2, T6, T10; Core 3: T3, T7, T11; Core 4: T4, T8, T12.
- **Dynamic Load Balancing (Least Loaded Scheduling)**: Tasks are assigned to the least loaded core at runtime.
- **Work Stealing**: Idle cores steal tasks from busy cores to balance the load.

### 1.1.6 Solution: Load Balancing

1. **Round-Robin Scheduling**:
   - Core 1: T1 (10 ms), T5 (30 ms), T9 (50 ms) = 90 ms
   - Core 2: T2 (15 ms), T6 (35 ms), T10 (55 ms) = 105 ms
   - Core 3: T3 (20 ms), T7 (40 ms), T11 (60 ms) = 120 ms
   - Core 4: T4 (25 ms), T8 (45 ms), T12 (65 ms) = 135 ms
2. **Overall Execution Time**: 135 ms (determined by the core with the longest execution time).

### 1.1.7 Cache and Memory Optimization

Optimizing cache and memory usage is critical for improving performance in multi-core systems. Key techniques include:

- **Cache Blocking**: Dividing data into smaller blocks that fit into the cache to reduce cache misses. For example, in matrix multiplication, the matrix can be divided into smaller sub-matrices that fit into the cache. This technique is particularly effective in scientific computing applications.
- **Prefetching**: Loading data into the cache before it is needed to reduce latency. For example, if a process is expected to access a specific memory location soon, the data can be prefetched into the cache. This is useful in applications with predictable memory access patterns.
- **False Sharing**: Avoiding unnecessary cache invalidations by aligning data structures to prevent multiple cores from accessing the same cache line. For example, if two threads frequently update different variables that reside on the same cache line, false sharing can occur, leading to performance degradation. This is a common issue in parallel programming.

### 1.1.8 Numerical Example: Cache and Memory Optimization

A matrix multiplication task with a cache miss rate of 40% is optimized using cache blocking and prefetching, reducing the execution time from 250 ms to 101.25 ms.

- **Cache Blocking**: The matrix is divided into 128x128 blocks to fit into the cache. This reduces the number of cache misses by ensuring that frequently accessed data remains in the cache.
- **Prefetching**: Data is loaded into the cache before it is needed, reducing cache misses. This is particularly effective in applications with predictable memory access patterns.
- **Execution Time Reduction**: From 250 ms to 101.25 ms.

### 1.1.9 Solution: Cache and Memory Optimization

1. **Initial Cache Miss Rate**: $CM_i = 40\%$
2. **Initial Execution Time**: $ET_i = 250$ ms
3. **Final Cache Miss Rate**: $CM_f = 18\%$
4. **Execution Time Reduction**:
$$ET_f = \frac{ET_i \times CM_f}{CM_i} = \frac{250 \times 0.18}{0.40} = 112.5 \text{ ms}$$
5. **Prefetching Reduction**: Further reduces execution time by 10%.
$$ET_{\text{new}} = ET_f \times (1 - 0.10) = 112.5 \times 0.90 = 101.25 \text{ ms}$$

# 2 Thread Libraries and Synchronization (2.2)

## 2.1 Thread Libraries

Thread libraries provide functionality for creating, managing, and synchronizing threads in multi-threaded applications. The most common thread libraries include:

- **POSIX Threads (pthreads)**: A standard for thread management in Unix-like systems. It provides functions for thread creation, synchronization, and management. pthreads is widely used in high-performance computing and real-time systems.
- **Windows Threads**: A threading library for Windows. It offers APIs for creating, suspending, resuming, and terminating threads. Windows Threads is commonly used in Windows-based applications and services.
- **OpenMP**: A high-level API for parallel programming. It simplifies parallel programming by providing compiler directives. OpenMP is popular in scientific computing and engineering applications.

## 2.2 Synchronization Mechanisms

### 2.2.1 Semaphores

Semaphores are used to control access to shared resources. They can be binary or counting semaphores.

- **Binary Semaphore (Mutex)**: Ensures mutual exclusion by allowing only one thread to access a critical section at a time. This is crucial in preventing race conditions and ensuring data consistency.
- **Counting Semaphore**: Manages access to a limited number of identical resources (e.g., database connections). This is useful in scenarios where multiple threads need to access a shared resource concurrently.

### 2.2.2 Conditional Variables

Conditional variables allow threads to wait for specific conditions to be met. They are commonly used in producer-consumer problems.

- **wait()**: Puts a thread to sleep until a condition is signaled. This is useful in scenarios where a thread needs to wait for a specific event to occur.
- **notify()**: Wakes up a single waiting thread. This is used to signal that a condition has been met and a waiting thread can proceed.
- **notify_all()**: Wakes up all waiting threads. This is useful in scenarios where multiple threads are waiting for the same condition.

#### 2.2.3 Synchronization Protocols

- **Mutex Locks**: Ensure mutual exclusion by allowing only one thread to access a critical section at a time. This is crucial in preventing race conditions and ensuring data consistency.
- **Readers-Writers Problem**: Allows multiple readers to access a resource simultaneously but requires exclusive access for writers. This is a common problem in database systems where read operations are more frequent than write operations.
- **Deadlock Prevention**: Protocols like wait-die and wound-wait prevent deadlocks by controlling how threads request resources. These protocols are essential in ensuring system stability and preventing resource contention.

## 2.3 Adaptive Synchronization Techniques

### 2.3.1 Adaptive Locks

Adaptive locks adjust their behavior based on the level of contention.

- **Spinlock**: A lock that repeatedly checks if it can acquire the lock and only blocks if it cannot. Useful in low-contention scenarios. Spinlocks are commonly used in real-time systems where threads need to acquire locks quickly.

### 2.3.2 Lock-Free Data Structures

Lock-free data structures avoid locking mechanisms to reduce contention and improve performance.

- **Compare-and-Swap (CAS)**: An atomic operation used to implement lock-free queues and stacks. CAS is a fundamental operation in lock-free programming and is used to ensure data consistency without locks.
- **Lock-Free Linked Lists**: Implemented using atomic operations to allow concurrent access without locks. This is useful in high-performance computing applications where lock contention can be a bottleneck.

### 2.3.3 Adaptive Synchronization Protocols

Adaptive synchronization protocols change their behavior based on the number of threads competing for a resource.

- **Queue-Based Locking**: Threads wait in a queue for their turn to acquire the lock, reducing contention. This is useful in scenarios where multiple threads need to access a shared resource concurrently.

- **Back-off Strategy**: Threads wait for a random period before retrying to acquire the lock, reducing contention. This is useful in high-contention scenarios where multiple threads are competing for the same resource.

## 3 Deadlock Detection Algorithms (2.3)

### 3.1 Centralized Deadlock Detection

In centralized deadlock detection, a central controller monitors the system for deadlocks using a wait-for graph.

- **Advantages**: Simple and efficient for small systems. Centralized deadlock detection is easy to implement and can quickly detect deadlocks in small-scale systems.
- **Disadvantages**: The central node becomes a bottleneck and a single point of failure. This can be a limitation in large-scale systems where the central node may become overwhelmed.

### 3.2 Distributed Deadlock Detection

In distributed systems, each process maintains a local wait-for graph and cooperates with other processes to detect deadlocks.

- **Advantages**: Scalable for large systems. Distributed deadlock detection can handle large-scale systems with many processes and resources.
- **Disadvantages**: Requires frequent communication between processes. This can introduce overhead and complexity in the system.

### 3.3 Hierarchical Deadlock Detection

In hierarchical deadlock detection, the system is divided into regions, each detecting deadlocks locally.

- **Advantages**: Efficient for large, hierarchical systems. Hierarchical deadlock detection can handle complex systems with multiple levels of resource management.
- **Disadvantages**: Complex to manage in dynamic environments. This can be a limitation in systems where resources and processes are frequently changing.

### 3.4 Example: Centralized Deadlock Detection

A system with three processes (P1, P2, P3) and two resources (R1, R2) detects a deadlock using a wait-for graph.

- **Processes and Resources**:
  - P1 holds R1 and waits for R2.
  - P2 holds R2 and waits for R1.
  - P3 waits for R1.
- **Wait-for Graph**:
  - P1 -¿ P2 (P1 waits for R2 held by P2)
  - P2 -¿ P1 (P2 waits for R1 held by P1)
  - P3 -¿ P1 (P3 waits for R1 held by P1)
- **Deadlock Detection**: A cycle in the wait-for graph (P1 -¿ P2 -¿ P1) indicates a deadlock.

## 3.5 Example: Distributed Deadlock Detection

In a distributed database system, nodes exchange messages to detect global deadlocks.

- **Nodes and Resources**:
  - Node 1 holds Resource A and waits for Resource B.
  - Node 2 holds Resource B and waits for Resource C.
  - Node 3 holds Resource C and waits for Resource A.
- **Message Exchange**:
  - Node 1 sends a request to Node 2 for Resource B.
  - Node 2 sends a request to Node 3 for Resource C.
  - Node 3 sends a request to Node 1 for Resource A.
- **Deadlock Detection**: A cycle in the global wait-for graph (Node 1 -¿ Node 2 -¿ Node 3 -¿ Node 1) indicates a deadlock.

## 3.6 Example: Hierarchical Deadlock Detection

In a cloud computing environment, local deadlocks are detected and reported to a higher-level coordinator.

- **Regions and Resources**:
  - Region 1: Process A holds Resource X and waits for Resource Y.
  - Region 2: Process B holds Resource Y and waits for Resource Z.
  - Region 3: Process C holds Resource Z and waits for Resource X.
- **Local Deadlock Detection**:

  - Region 1 detects a local deadlock (Process A -¿ Process B).
  - Region 2 detects a local deadlock (Process B -¿ Process C).
  - Region 3 detects a local deadlock (Process C -¿ Process A).

- **Global Deadlock Detection**: The higher-level coordinator detects a global deadlock (Region 1 -¿ Region 2 -¿ Region 3 -¿ Region 1).

## 4 Distributed Mutual Exclusion Algorithms (2.4)

### 4.1 Lamport's Algorithm

Lamport's algorithm uses logical timestamps to ensure mutual exclusion in distributed systems.

- **Logical Timestamps**: Each process maintains a logical clock that is incremented with each event. The logical clock is used to order events in the system.
- **Request and Reply Messages**: When a process wants to enter the critical section, it sends a request message to all other processes. The process can enter the critical section only after receiving replies from all other processes.
- **Mutual Exclusion**: Lamport's algorithm ensures that only one process can enter the critical section at a time, preventing conflicts and ensuring data consistency.

### 4.2 Token-Based Algorithms

- **Ricart-Agrawala**: Uses request and reply messages to ensure mutual exclusion. Each process sends a request message to all other processes and can enter the critical section only after receiving replies from all other processes.
- **Maekawa's Algorithm**: Uses voting sets to ensure mutual exclusion. Each process has a voting set of neighboring processes. A process can enter the critical section only after receiving permission from all processes in its voting set.

### 4.3 Epidemic Algorithms

Epidemic algorithms use probabilistic methods to achieve mutual exclusion in distributed systems.

- **Probabilistic Methods**: Epidemic algorithms use randomized techniques to propagate information through the system. This approach is robust and can handle dynamic changes in the system.

- **Mutual Exclusion**: Epidemic algorithms ensure that only one process can enter the critical section at a time, preventing conflicts and ensuring data consistency.

## 5 Case Studies (2.5)

### 5.1 Process Management in Linux

Linux uses a combination of affinity-based scheduling, load balancing, and NUMA-aware scheduling to optimize process management.

- **Affinity-Based Scheduling**: Linux uses CPU affinity to bind processes to specific CPU cores, reducing context switching and improving cache locality.
- **Load Balancing**: Linux employs dynamic load balancing techniques to distribute tasks evenly across CPU cores, preventing overloading and underutilization.
- **NUMA-Aware Scheduling**: Linux optimizes memory access by allocating memory close to the CPU core where the process runs, minimizing memory access latency.

### 5.2 Process Management in Windows

Windows employs priority-based scheduling and dynamic load balancing to manage processes efficiently.

- **Priority-Based Scheduling**: Windows uses a priority-based scheduling algorithm to ensure that high-priority processes receive more CPU time than low-priority processes.
- **Dynamic Load Balancing**: Windows dynamically assigns tasks to CPU cores based on the current load, ensuring efficient utilization of resources.

## Conclusion

Advanced process and thread management techniques are essential for optimizing performance in modern multi-core and distributed systems. By leveraging affinity-based scheduling, load balancing, cache optimization, and deadlock detection algorithms, systems can achieve high efficiency and reliability. These techniques are crucial in ensuring that modern computing systems can handle complex workloads and provide consistent performance.