Exam 3 Notes

CPU Scheduling Overview

CPU scheduling is essential for managing how processes are assigned to the CPU. The main goal is to optimize various performance metrics, such as response time, wait time, and throughput, by selecting which process should run next. Different scheduling algorithms prioritize tasks based on specific criteria, balancing efficiency and fairness among processes.

Schedulers determine how CPU time is distributed, and various strategies impact how long processes wait and how quickly they are completed. Understanding these strategies is critical for improving system performance, especially in multi-tasking environments.

Response Time and Wait Time

- **Response Time**: The time from a process's first entry into the ready queue to its first scheduling on the CPU.
- **Wait Time**: The sum of gaps between time slices given to a process while it waits to execute on the CPU.

CPU Scheduling Algorithms

Different scheduling algorithms are used to optimize various aspects of CPU performance. These include First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Earliest Deadline First (EDF). Each has its strengths and trade-offs depending on the system's requirements.

Scheduling Algorithms

- **FCFS**: Simple but can lead to long wait times for processes arriving later.
- **SJF**: Reduces average wait time by prioritizing shorter processes, providing better overall performance.
- **RR**: Allocates CPU time slices to processes, cycling through them to ensure all receive processing time.

Multi-Level Feedback Queue Scheduling

Multi-level feedback queues are more flexible than simple multi-level queue scheduling. They allow processes to move between queues based on their CPU usage, ensuring dynamic adjustments in priority based on their behavior.

Multi-Level Feedback Queues

- **Dynamic Priority**: Processes move between priority queues based on their CPU usage, providing better adaptability.
- **Efficiency**: Prevents starvation by allowing lower-priority processes to eventually receive CPU time.

Completely Fair Scheduler (CFS)

The Completely Fair Scheduler (CFS) used in Linux systems focuses on fairness by using a virtual runtime (vruntime) to prioritize processes. Instead of actual run time, vruntime accounts for CPU usage, giving processes that use less CPU more favorable scheduling.

Completely Fair Scheduler (CFS)

- **vruntime**: Ensures fairness by favoring processes that have used less CPU time.
- **Red-Black Tree**: CFS uses a self-balancing red-black tree to select the next process, ensuring efficient scheduling.

Symmetric Multi-Processing (SMP)

In symmetric multi-processing (SMP) systems, all CPU cores are self-scheduling, and each can execute processes independently. This model balances the load across all processors and optimizes overall performance by reducing CPU idle time.

Symmetric Multi-Processing (SMP)

- **Self-Scheduling Cores**: Each CPU core schedules itself, contributing to more efficient process management.
- **Load Balancing**: Distributes processes evenly across CPU cores to avoid overloading any single core.

Key Concepts Summary

- **Response and Wait Times**: Crucial for measuring scheduling effectiveness.
- **Scheduling Algorithms**: SJF reduces wait times, while RR ensures fairness.
- **Multi-Level Feedback Queues**: Adjust priorities dynamically based on CPU usage.
- **CFS**: Focuses on fairness using vruntime and red-black tree structures.
- **SMP**: Balances the workload across CPU cores to enhance multi-processing performance.

CPU scheduling techniques balance efficiency, fairness, and system performance. Choosing the right algorithm depends on system requirements, process behavior, and hardware capabilities.

Process Synchronization Overview

Process synchronization ensures orderly execution when multiple threads or processes share resources. Synchronization mechanisms prevent race conditions, where concurrent processes attempt to modify shared data simultaneously, leading to unpredictable outcomes. Critical sections in the code are areas where processes access shared resources, and synchronization tools ensure only one process can access these resources at a time.

Understanding synchronization is essential for safe and efficient multi-threaded applications, especially when handling shared resources that could otherwise lead to data corruption or deadlock.

Key Terms in Synchronization

- **Race Condition**: Occurs when multiple threads attempt to modify shared data concurrently.
- **Concurrency**: Execution of multiple processes to simulate parallelism.
- **Synchronization**: Mechanisms to control the order of execution in concurrent systems.
- **Critical Section**: Code section where shared resources are accessed.
- **Mutual Exclusion**: Prevents more than one process from accessing a critical section simultaneously.

Atomic Operations and Test-and-Set

Atomic operations execute as a single, indivisible step, critical for ensuring synchronization without interference. The Test-and-Set operation is one example, providing mutual exclusion by atomically setting a variable and returning its old value. However, Test-and-Set does not block processes and thus is unsuitable for directly handling mutual exclusion under all circumstances.

Test-and-Set Properties

- **Atomic Operation**: Executes without interference from other processes.
- **Mutex Mechanism**: Ensures mutual exclusion, but does not block processes.

Condition Variables and Monitors

Condition variables provide synchronization by allowing threads to wait for certain conditions before proceeding. They keep track of waiting threads and impose an ordering on thread access. Monitors are high-level constructs that bundle condition variables with mutual exclusion, providing a structured way to handle synchronization.

Condition Variables and Monitors

- **Condition Variables**: Block threads until specific conditions are met and manage thread access order.
- **Monitors**: Implicitly provide mutual exclusion, allowing only one thread to execute within the monitor at any time.

Mutexes and Semaphores

Mutexes and semaphores are used to manage access to shared resources. A binary semaphore behaves like a mutex, allowing mutual exclusion for a single resource, while counting semaphores handle multiple resources. Semaphore operations, typically called P() and V(), are equivalent to wait() and signal(), controlling access to resources in an atomic manner.

Comparison of Mutexes and Semaphores

- **Mutex vs Binary Semaphore**: Both provide mutual exclusion, but semaphores can manage multiple threads.
- **Counting Semaphore**: Tracks the availability of multiple resources, unlike mutexes, which are binary.
- **P() and V() Operations**: Also known as wait() and signal(), used for atomic resource control.

Deadlocks and Semaphore Challenges

Deadlock is a potential problem with semaphores, occurring when processes are stuck waiting indefinitely for resources. This can happen if synchronization is improperly implemented, allowing processes to lock resources in an unsolvable cycle. Effective use of semaphores and careful resource management can mitigate the risk of deadlocks.

Deadlock in Synchronization

- **Deadlock Risk**: Processes can be stuck waiting for resources held by each other, creating a deadlock cycle.
- **Avoiding Deadlock**: Implement careful resource allocation and release practices to prevent deadlock.

Summary of Key Concepts

- **Race Conditions**: Occur when multiple threads modify shared data simultaneously without proper synchronization.
- **Condition Variables and Monitors**: Manage thread access to resources, preventing race conditions.
- **Mutexes vs Semaphores**: Mutexes are simpler, while semaphores allow complex resource management.
- **Deadlock Prevention**: Essential to ensure processes do not lock resources in an unsolvable cycle.

Process synchronization tools like mutexes, semaphores, and condition variables are critical for maintaining data integrity and ensuring smooth operation in concurrent systems. Properly implemented, they prevent race conditions and deadlocks, enhancing system reliability and efficiency.

Deadlock Concepts Overview

Deadlocks occur when processes are permanently blocked because they are each holding a resource and waiting for resources held by other processes. Four necessary and sufficient conditions must hold simultaneously for deadlock to occur: mutual exclusion, hold-and-wait, no preemption, and circular wait. Deadlock prevention, avoidance, and detection are strategies to manage deadlocks, ensuring system resources are effectively allocated without causing indefinite blocking.

Properly managing resources in a multi-process environment is essential to prevent deadlocks and to maintain system stability.

Necessary Conditions for Deadlock

- **Mutual Exclusion**: At least one resource must be held in a non-shareable mode.
- **Hold-and-Wait**: Processes must hold at least one resource and wait to acquire additional resources.
- **No Preemption**: Resources cannot be forcibly removed; they must be released voluntarily.
- **Circular Wait**: A circular chain of processes exists, each waiting for a resource held by the next.

Deadlock Prevention, Avoidance, and Detection

Deadlock management includes three main strategies. Prevention techniques alter the system's conditions to ensure that at least one of the four necessary deadlock conditions cannot hold. Deadlock avoidance uses algorithms, such as the Banker's algorithm, to ensure the system remains in a safe state by checking if a resource request might lead to an unsafe state. Detection allows deadlocks to occur and periodically checks for them to resolve any cycles of waiting processes.

Deadlock Strategies

- **Deadlock Prevention**: Ensures at least one deadlock condition does not hold.
- **Deadlock Avoidance**: Dynamically checks requests using algorithms like Banker's to avoid unsafe states.
- **Deadlock Detection**: Allows deadlocks but detects them periodically for correction.

Banker's Algorithm

The Banker's algorithm is a deadlock avoidance method that evaluates each resource request to determine if fulfilling it would leave the system in a safe state. By simulating allocations, the algorithm ensures there are enough resources for all processes to complete without leading to deadlock.

Banker's Algorithm

- **Safe State**: The system can allocate resources such that all processes can eventually complete.
- **Resource Allocation Simulation**: Determines if granting a request maintains a safe state.

Common Deadlock Prevention Techniques

Preventing deadlock can involve numbering resources and requesting them in a specific order or ensuring a process releases all resources before requesting new ones. By carefully managing resource requests and releases, the system avoids conditions that lead to circular waits and resource contention.

Deadlock Prevention Techniques

- **Ordered Resource Requests**: Processes only request resources in a predetermined order.
- **Resource Release Before Request**: Processes must release all resources before requesting new ones.

Summary of Key Concepts

- **Necessary Deadlock Conditions**: Mutual exclusion, hold-and-wait, no preemption, and circular wait.
- **Deadlock Prevention, Avoidance, Detection**: Strategies to manage resource allocation and prevent indefinite blocking.
- **Banker's Algorithm**: A method to ensure safe resource allocation by simulating requests.
- **Prevention Techniques**: Ordered requests and resource release practices reduce deadlock risks.

Deadlock management is critical in multi-process systems to maintain efficient resource allocation and prevent processes from becoming permanently blocked.