Synchronization Concepts and Best Practices

Overview

Processes Synchronization or Synchronization is the way by which processes that share the same memory space are managed in an operating system. It helps maintain the consistency of data by using variables or hardware so that only one process can make changes to the shared memory at a time. There are various solutions for the same such as semaphores, mutex locks, synchronization hardware, etc.

What is Process Synchronization in OS?

Process synchronization in an operating system ensures that multiple processes that share a common resource do not interfere with each other, leading to inconsistent data. For example, if two processes simultaneously access and modify a shared resource, like a bank account balance, incorrect or inconsistent results can occur. Synchronization helps to prevent such conflicts by ensuring that only one process accesses the shared resource at a time, thereby maintaining data consistency.

Key Synchronization Concepts

Process synchronization?

Process synchronization is the process of coordination or multiple processes or threads that share common resources to execute and prevent conflicts between the processes or do not interfere with each other while sharing resources and ensuring data consistency. It's essential in multi-threaded or multi-process systems where concurrent access to shared resources can lead to issues like race conditions , deadlocks, and inconsistencies.

Semaphores

A semaphore is a synchronization mechanism used in computer science to manage access to shared resources, ensuring that multiple processes or threads do not interfere with each other, especially in critical sections of code. Critical sections are code segments where shared resources like memory or I/O devices are accessed, and it's essential to allow only one process to execute at a time to prevent issues like race conditions.

Semaphores are classified as **synchronization primitives** and can be used to implement **mutual exclusion** and **process synchronization**. The two key operations of a semaphore are:

1. Wait (P): This operation checks the semaphore's current value. If the value is greater than zero, it decrements the value, allowing the process to proceed. If the value is zero, the process is blocked and must wait until another process releases the resource by performing a signal operation.

2. Signal (V): This operation increments the semaphore's value, essentially releasing the resource. If there are any processes blocked due to a previous wait operation, one of them is unblocked and allowed to proceed.

Semaphores help prevent **race conditions**, which occur when processes access shared resources simultaneously in a way that leads to unpredictable or incorrect outcomes. They ensure mutual exclusion, meaning only one process accesses the shared resource at a time.

Semaphores are also known as **counting semaphores** because their value reflects how many processes can successfully execute the wait operation before getting blocked. The initial value of a semaphore sets the limit on how many processes can enter the critical section simultaneously.

Mutual Exclusion in Synchronization

Mutual Exclusion is a property of process synchronization that states that "**no two processes can exist in the critical section at any given point of time**". The term was first coined by **Dijkstra**. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

The need for mutual exclusion comes with concurrency. There are several kinds of concurrent execution:

- Interrupt handlers
- Interleaved, preemptively scheduled processes/threads
- Multiprocessor clusters, with shared memory
- Distributed systems

Mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

The requirement of mutual exclusion is that when process P1 is accessing a shared resource R1, another process should not be able to access resource R1 until process P1 has finished its operation with resource R1.

Conditions Required for Mutual Exclusion

According to the following four criteria, mutual exclusion is applicable:

- When using shared resources, it is important to ensure mutual exclusion between various processes. There cannot be two processes running simultaneously in either of their critical sections.
- It is not advisable to make assumptions about the relative speeds of the unstable processes.
- For access to the critical section, a process that is outside of it must not obstruct another process.
- Its critical section must be accessible by multiple processes in a finite amount of time; multiple processes should never be kept waiting in an infinite loop.

Key Concepts

- **Race Conditions:** It occurs when some common shared resources are accessed by multiple processes or threads, leading to unpredictable behavior.
- **Critical Section:** section of code where shared resources are accessed. Only one process or thread should be allowed to execute the critical section simultaneously.
- **Deadlocks:** a situation occurs when two or more processes are holding the resources and do not release them causing a deadlock to occur because another is waiting for that resources. Deadlock prevention and detection mechanisms are critical in process synchronization.
- **Starvation:** One process is being unable to access resources due to other processes holding onto them for extended periods, often due to priority scheduling or unfair resource allocation.

Synchronization Techniques Include

- **Mutex(Mutual Exclusion):** only one process can access the critical section at a time, it is a locking mechanism.
- **Semaphores:** Signaling mechanisms to control access to shared resources.
- **Monitors:** High-level synchronization constructs that encapsulate shared resources and provide a safe interface for processes to access them.
- **Locks** : Mechanisms to protect shared resources from concurrent access. Locks can be simple or complex.

Lock Variable Synchronization Mechanism

A lock variable provides the simplest synchronization mechanism for processes. Some noteworthy points regarding Lock Variables are-

1. It’s a **software mechanism** implemented in user mode, i.e. no support required from the Operating System.
2. It’s a busy waiting solution (keeps the CPU busy even when its technically waiting).
3. It can be used for more than two processes.

Now every Synchronization mechanism is judged on the basis of three primary parameters

1. Mutual Exclusion.
2. Progress.
3. Bounded Waiting.

Of which mutual exclusion is the most **important** of all parameters. The Lock Variable doesn’t provide mutual exclusion in some cases.

Best Practices for synchronization

- Use synchronized blocks when you want to synchronize a part of the method rather than the entire method.
- Synchronized blocks can be more efficient as they allow threads to execute non-synchronized parts of a method concurrently
- Minimize Critical Sections: Keep the code within critical sections as short as possible to reduce the chance of contention and improve performance.

- Use High-Level Concurrency Libraries: Utilize built-in libraries or frameworks that handle synchronization, as they are often more reliable and optimized.
- Avoid Deadlocks: Implement strategies such as locking hierarchy or timeout-based locking to prevent deadlocks.
- Test Thoroughly: Perform stress tests and use debugging tools to identify and fix synchronization issues.
- Document and Review: Clearly document synchronization logic and regularly review it to ensure it's still effective and relevant.

REFERENCES