Table of Contents

[Synchronization 2](#_Toc135858332)

[Producer-Consumer Problem 2](#_Toc135858333)

[Race Condition 2](#_Toc135858334)

[The Critical Section Problem 3](#_Toc135858335)

[Solution to Critical-Section Problem 3](#_Toc135858336)

[Critical-Section Handling in OS 3](#_Toc135858337)

[Software Solutions: 4](#_Toc135858338)

[Peterson’s Algorithm 4](#_Toc135858339)

[Mutex Locks 4](#_Toc135858340)

[Semaphore 4](#_Toc135858341)

[Monitor 5](#_Toc135858342)

[SYNCHRONIZATION HARDWARE 5](#_Toc135858343)

# Synchronization

Synchronization is a technique used in computer programming to coordinate the access of multiple processes to shared resources, such as files or variables. If multiple processes try to access a shared resource at the same time, it can lead to errors or inconsistencies in the data. To avoid this, synchronization mechanisms are used to ensure that only one process accesses the resource at a time. Atomic instructions are used to ensure that the instructions are executed as a single unit and cannot be interrupted, ensuring data consistency and avoiding errors.

## Producer-Consumer Problem

The producer-consumer problem is a challenge that happens when multiple processes need to produce and consume data from a shared buffer. We need to ensure that they don't access the same buffer at the same time, or else errors can occur.

One solution to this problem is to use a shared variable that is only changed by one process at a time. However, this solution has a limitation - it can only use BUFFER\_SIZE - 1 elements.

To overcome this limitation and fill all the buffers, we can use a more complex solution. We introduce another variable called counter to keep track of the number of full buffers. The producer process increments the value of the "in" variable and the counter after producing a new buffer. The consumer process increments the value of the "out" variable and decrements the counter after consuming a buffer. This way, we can ensure that only one process accesses a buffer at a time and all buffers are used.

## Race Condition

If two or more processes try to change the same variable (like counter) at the same time, it can lead to errors because the outcome depends on the order in which they access the variable. This situation is called a race condition. To prevent this, we need to ensure that only one process can change the variable at a time.

# The Critical Section Problem

In a system of multiple processes, each process may have a critical section of code where it is accessing or modifying shared data. However, to prevent conflicts and data inconsistencies, only one process should be allowed in the critical section at a time. This problem is known as the critical section problem, and it requires a protocol to ensure that only one process enters the critical section at a time. The protocol involves a process asking for permission to enter the critical section, execute in it, and then exit it. However, this may result in "busy waiting" if a process is waiting for another to finish executing in the critical section.

## Solution to Critical-Section Problem

Mutual Exclusion: Only one process can be executing in the critical section at any time.

Progress: If no process is in the critical section and one or more processes want to enter it, then a process must be chosen to enter the critical section soon.

Bounded Waiting: After a process requests access to the critical section, there is a limit on the number of times other processes can enter their critical sections before the requesting process is allowed to enter.

Or

Bounded Waiting: There must be a limit to the number of times other processes can enter the critical section before a waiting process enters.

## Critical-Section Handling in OS

Do yourself.

<https://sphere-labs.notion.site/FGenEds-Cheat-Sheet-NewCh5-pdf-bef89e5637194fd18643d1744c665c77>

# Software Solutions:

## Peterson’s Algorithm

Peterson's algorithm is a software solution used to solve the critical section problem for two processes. The algorithm assumes that the load and store machine-language instructions are atomic and cannot be interrupted. The two processes share two variables: 'turn' and a Boolean array 'flag' of size 2. The 'turn' variable is used to indicate whose turn it is to enter the critical section, while the 'flag' array indicates if a process is ready to enter the critical section. The algorithm guarantees that mutual exclusion, progress, and bounded-waiting requirements are met. However, this algorithm is only useful for two processes and cannot be extended to N > 2 processes.

## Mutex Locks

To solve the critical section problem, operating system designers have created software tools that are accessible to application programmers. One of the simplest tools is called a mutex lock. A mutex lock is used to protect a critical section of code by acquiring the lock before entering the section and releasing it afterwards. It uses a Boolean variable to indicate whether the lock is available or not. The functions to acquire and release the lock must be atomic, meaning they should not be interrupted. This is typically achieved using hardware atomic instructions. However, this solution requires busy waiting, where a thread continuously checks if the lock is available. Due to this characteristic, this type of lock is commonly referred to as a spinlock.

## Semaphore

Semaphore is a software solution for the critical section problem. It is a variable that controls access to a shared resource. It supports "wait" and "signal" operations. Processes "wait" for the semaphore if the resource is not available and "signal" to release it when they are done. Semaphores provide flexible synchronization for multiple resources but require careful usage to avoid issues like deadlock or race conditions.

## Monitor

Monitor is a software solution for the critical section problem that combines data and code into a single construct. It provides mutual exclusion by allowing only one process to execute inside the monitor at a time. Other processes are queued and wait for their turn.

# SYNCHRONIZATION HARDWARE

Synchronization hardware is a set of instructions that can be used to solve the critical section problem. On uniprocessors, this can be done by disabling interrupts, which prevents other processes from running while the critical section code is executing. However, this is not efficient on multiprocessors, and operating systems that use this method are not scalable. Modern machines provide special atomic hardware instructions that can be used to solve the critical section problem without disabling interrupts. These instructions are either test-and-set or swap.