# /Introduction of Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:

* **Independent Process**: The execution of one process does not affect the execution of other processes.
* **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

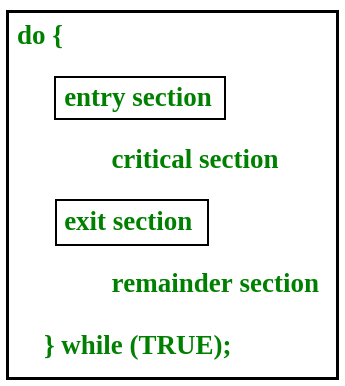
Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

### Race Condition:

 When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

### Critical Section Problem:

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

[](https://www.geeksforgeeks.org/wp-content/uploads/gq/2015/06/critical-section-problem.png)

**What is Semaphore?**

If you have read about Process Synchronization, you are aware of the critical section problem that arises for concurrent processes.

If not, let's quickly get comfortable with these terms above.

Concurrent Processes are those processes that are executed simultaneously or parallely and might or might not be dependent on other processes. Process Synchronization can be defined as the coordination between two process that have access to common materials such as a common section of code, resources or data etc. **For example:** There may be some resource that is shared by 3 different processes, and none of the processes at a certain time can change the resource, since that might ruin the results of the other processes sharing the same resource. You'll understand it more clearly soon.

Now this Process Synchronization is required for concurrent processes. For any number of processes that are executing simultaneously, let's say all of them need to access a section of the code. This section is called the Critical Section.

Now that you are familiar with these terms, we can move on to understanding the need of Semaphores with an example.

We have 2 processes, that are concurrent and since we are talking about Process Synchronization, let's say they share a variable "shared" which has a value of 5. What is our goal here? We want to achieve mutual exclusion, meaning that we want to prevent simultaneous access to a shared reource. The resource here being the variable "shared" with value 5.

int shared = 5

**Process 1**

int x = shared; *// storing the value of shared variable in the variable x*

x++;

sleep(1);

shared = x;

**Process 2**

int y = shared;

y--;

sleep(1);

shared = y;

We start with the execution of process 1, in which we declare a variable x which has initially the value of the shared variable which is 5. The value of x is then incremented, and it becomes 6 and post that the process goes into sleep state. Since the current processing is concurrent, the cpu does not wait and starts the processing of process 2. The integer y has the value of the shared variable initially which is unchanged, and is 5.

Then we decrement the value of y and process 2 goes into sleep state. We move back to process 1 and the value of shared variable becomes 6. Once that process is complete, in process 2 the value of shared variable is changed to 4.

One would think that if we increment and decrement a number, it's value should be unchanged and that is exactly what was happening in the two processes, however in this case the value of the "shared" variable is 4, and this is undesired.

**For example:** If we have 5 resources, and one process uses it, decrementing it's value by 1, just like in our example -- process X, had done. And if another process Y releases the same resource it had taken earlier, a similar situation might occur and the resultant would be 4, which instead should have been 5 itself.

This is called a ***race condition***, and due to this condition, problems such as deadlock may occur. Hence we need proper synchronization between processes, and to prevent these, we use a signaling integer variable, called - Semaphore.

So to formally define Semaphore we can say that it is an integer variable which is used in a mutually exclusive manner by concurrent processes, to achieve synchronization.

Since **Semaphores are integer variables, their value acts as a signal, which allows or does not allow a process to access the critical section of code or certain other resources**.

**Types of Semaphores**

There are mainly two types of Semaphores, or two types of signaling integer variables:

**1. Binary Semaphores**

In these type of Semaphores the integer value of the semaphore can only be either 0 or 1. If the value of the Semaphore is 1, it means that the process can proceed to the critical section (the common section that the processes need to access). However, **if the value of binary semaphore is 0, then the process cannot continue to the critical section of the code**. When a process is using the critical section of the code, we change the Semaphore value to 0, and when a process is not using it, or we can allow a process to access the critical section, we change the value of semaphore to 1. **Binary semaphore is also called mutex lock**.We'll discuss the working of the semaphores soon.

**2. Counting Semaphores**

**Counting semaphores** are signaling integers that can take on any integer value. Using these Semaphores we can coordinate access to resources and here the Semaphore count is the number of resources available. If the value of the Semaphore is anywhere above 0, processes can access the critical section, or the shared resources. The number of processes that can access the resources / code is the value of the semaphore. However, if the value is 0, it means that there aren't any resources that are available or the critical section is already being accessed by a number of processes and cannot be accessed by more processes. **Counting semaphores are generally used when the number of instances of a resource are more than 1, and multiple processes can access the resource**.

**Example of Semaphore**

Now that we know what semaphores are and and their types, we must understand their working. As we read above, **our goal is to synchronize processs and provide mutual exclusion in the critical section of our code**. So, we have to introduce a mechanism that wouldn't allow more than 1 process to access the critical section using the signaling integer - semaphore.

shared variable semaphore = 1;

process i

begin

.

.

P(mutex);

execute Critical Section

V(mutex);

.

.

end;

Here in this piece of pseudocode, we have declared a semaphore in line 1, which has the value of 1 initially. We then start the execution of a process i which has some code, and then as you can see, we call a function "P" which takes the value of mutex/semaphore as input and we then proceed to the critical section, followed by a function "V" which also takes the value of mutex / semaphore as input. Post that, we execute the remainder of the code, and the process ends.

Remember, we discussed that **semaphore is a signaling variable and whether or not the process can proceed to the critical section depends on its value**. And in binary and counting semaphores we read that we change the value of the semaphore according to the reources available. With this thought, let's move further to read about these "P" and "V" functions in the above pseudocode.

**Problem Statement –** We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

**Initialization of semaphores –**   
mutex = 1   
Full = 0 // Initially, all slots are empty. Thus full slots are 0   
Empty = n // All slots are empty initially

**Solution for Producer –**

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

**Solution for Consumer –**

do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.