V PROCESS SYNCHRONIZATION

- The concept of Process synchronization in an Operating System.
- Process Synchronization was introduced to handle problems that arosewhile multiple process executions.
- Process is categorized into two types on the basis of synchronization andthese are given below:
  - Independent Process
  - Cooperative Process

1.Independent Processes

Two processes are said to be independent if the execution of one processdoes not affect the execution of another process.

2.Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization

- It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.
- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-processsystem when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

1. Race Condition

At the time when more than one process is either 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 purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as **a race condition.** As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the accessof data takes place.

Mainly this condition is a situation that may occur inside the **critical section**. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

2. Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry tothe critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.

3. Entry Section

In this section mainly the process requests for its entry in thecritical section.

4. Exit Section

This section is followed by the critical section.

The solution to the Critical Section Problem

A solution to the critical section problem must satisfy the followingthree conditions:

1. *Mutual Exclusion*

Out of a group of cooperating processes, only one process can be in itscritical section at a given point of time.

2. *Progress*

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. *Bounded Waiting*

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

Solutions for the Critical Section

The critical section plays an important role in Process Synchronization so that the problem must be solved. Some widely used method to solve the critical section problem is asfollows:

1. **Peterson's Solution**

- This is widely used and software-based solution to critical section problems. Peterson's solution was developed by a computer scientist Peterson that's why it is named so.
- With the help of this solution whenever a process is executing in any critical state, then the other process only executes the rest of the code, and vice-versa can happen. This method also helps to make sure of the thing that only a single process can run in the critical section at a specific time.

This solution preserves all three conditions:

- Mutual Exclusion is comforted as at any time only one process can access the critical section.
- Progress is also comforted, as a process that is outside the critical section is unable to block other processes from entering into the critical section.
- Bounded Waiting is assured as every process gets a fair chance to enter the Critical section.
- Suppose there are **N processes (P1, P2, ... PN)** and as at some pointof time every process requires to enter in the **Critical Section**
- A **FLAG[]** array of size N is maintained here which is by defaultfalse. Whenever a process requires to enter in the critical section, it has to set its flag as true. Example: If Pi wants to enter it will set **FLAG[i]=TRUE.**
- Another variable is called **TURN** and is used to indicate the process number that is currently waiting to enter into the critical section.
- The process that enters into the critical section while exiting would change the **TURN** to another number from the list of processes that are ready.
- Example: If the turn is 3 then P3 enters the Critical section andwhile exiting turn=4 and therefore P4 breaks out of the wait loop.

Synchronization Hardware

- Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.
- Disabling interrupt on a multiprocessor environment can be time-consuming as the message is passed to all the processors.
- This message transmission lag delays the entry of threads into thecritical section, and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside the criticalsection, and in the exit section that LOCK is released.