Introduction of Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

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 processes 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, and 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.

Example:

Let's understand one example to understand the race condition better:

Let's say there are two processes P1 and P2 which share a common variable (shared=10), both processes are present in – queue and waiting for their turn to be executed. Suppose, Process P1 first come under execution, and the CPU store a common variable between them (shared=10) in the local variable (X=10) and

increment it by 1(X=11), after then when the CPU read line sleep(1),it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in a waiting state for 1 second.
Now CPU execute the Process P2 line by line and store common variable (Shared=10) in its local variable (Y=10) and decrement Y by 1(Y=9), after then when CPU read sleep(1), the current process P2 goes in waiting for state and CPU remains idle for some time as there is no process in ready-queue, after completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code (store the local variable (X=11) in common variable (shared=11)), CPU remain idle for sometime waiting for any process in ready-queue,after completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2(store the local variable (Y=9) in common variable (shared=9)).

Initially Shared = 10

Process 1	Process 2
int X = shared	int Y = shared
X++	Y–
sleep(1)	sleep(1)
shared = X	shared = Y

Note: We are assuming the final value of a common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable (shared=10) by 1 and Process P2 decrement variable (shared=11) by 1 and finally it becomes shared=10). But we are getting undesired value due to a lack of proper synchronization.

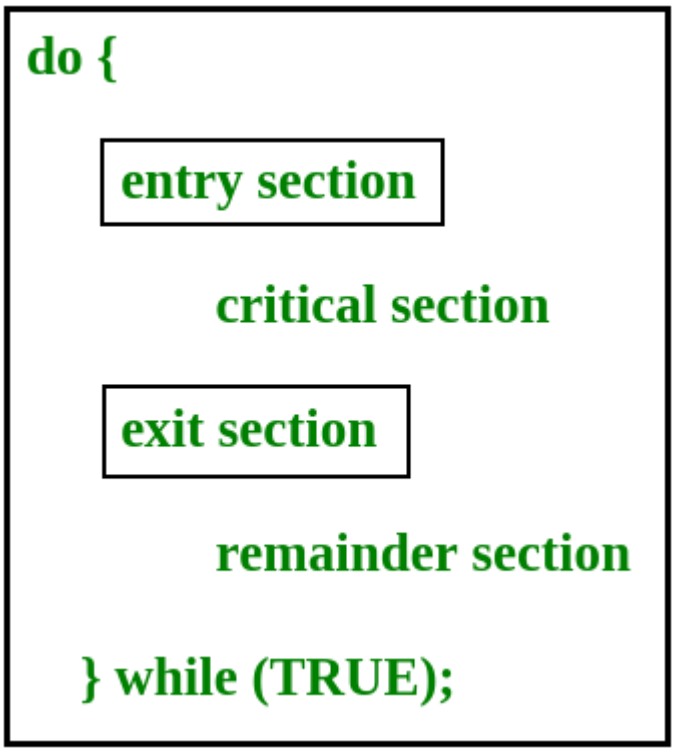
Actual meaning of race-condition

- If the order of execution of the process(first P1 -> then P2) then we will get the value of common variable (shared) =9.
- If the order of execution of the process(first P2 -> then P1) then we will get the final value of common variable (shared) =11.
- Here the (value1 = 9) and (value2=10) are racing, If we execute these two processes in our computer system then sometime we will get 9 and sometime we will get 10 as the final value of a common variable(shared). This phenomenon is called race condition.

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.



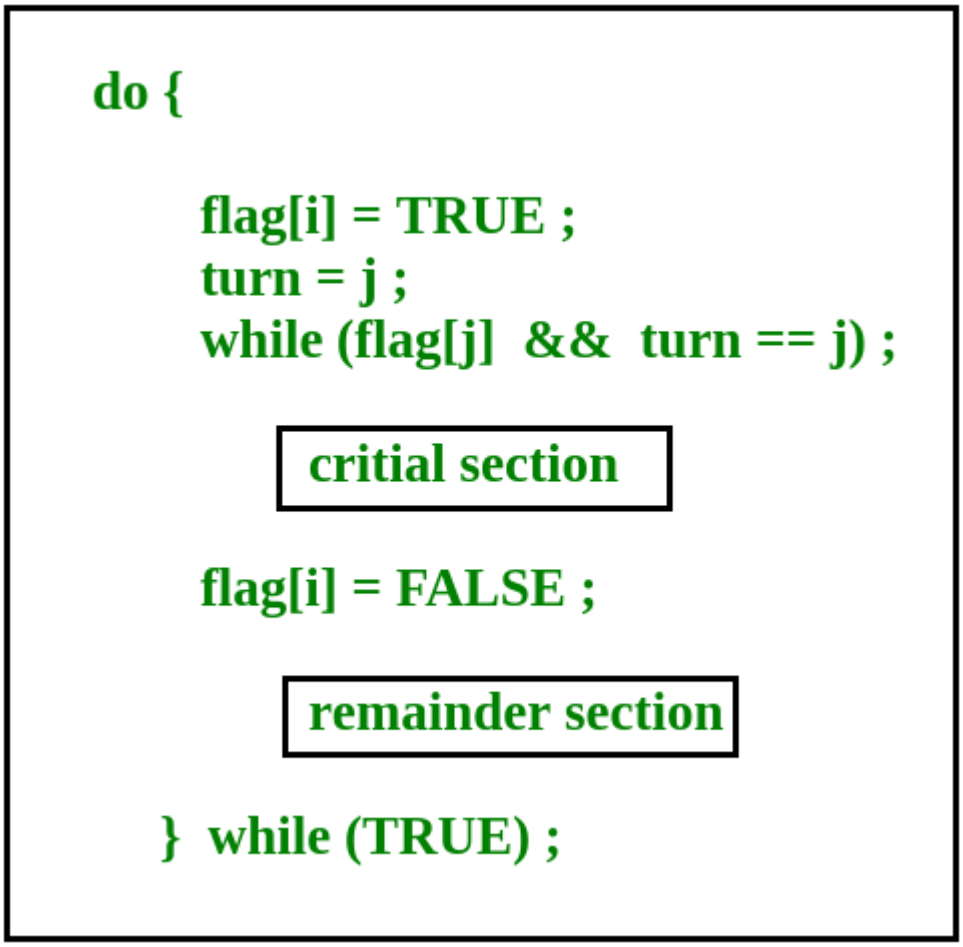
In the entry section, the process requests for entry in the **Critical Section.**
Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.



Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves busy waiting. (In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

Semaphores

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that is called the wait function. A semaphore uses two atomic operations, wait and signal for process synchronization. A Semaphore is an integer variable, which can be accessed only through two operations :

- wait() and signal().
  There are two types of semaphores: Binary Semaphores and Counting Semaphores.
  - **Binary Semaphores:** They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.
  - **Counting Semaphores:** They can have any value and are not restricted to a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Advantages of Process Synchronization

- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

Disadvantages of Process Synchronization

- Adds overhead to the system
- This can lead to performance degradation
- Increases the complexity of the system
- Can cause deadlocks if not implemented properly.