Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multiprocess 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.

What is Process?

A process is a program that is currently running or a program under execution is called a process. It includes the program's code and all the activity it needs to perform its tasks, such as using the CPU, memory, and other resources. Think of a process as a task that the computer is working on, like opening a web browser or playing a video.

Types of Process

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.

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 deadlock if not implemented properly.

Race Conditions

What is 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 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

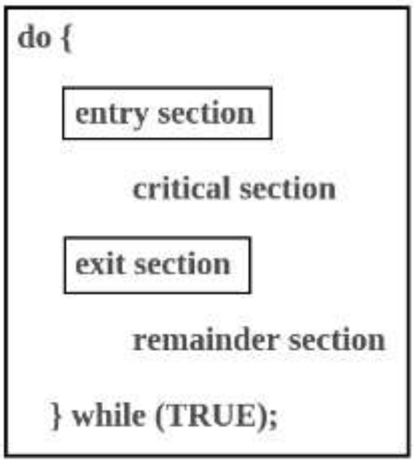
Note: We are assuming the final value of a common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable (shared=10) by 1 and Process P2 decrement variable (shared=11) by 1 and finally it becomes shared=10). But we are getting undesired value due to a lack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



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

In the entry section, the process requests for entry in the **Critical Section.**

Any solution to the critical section problem must satisfy three requirements:

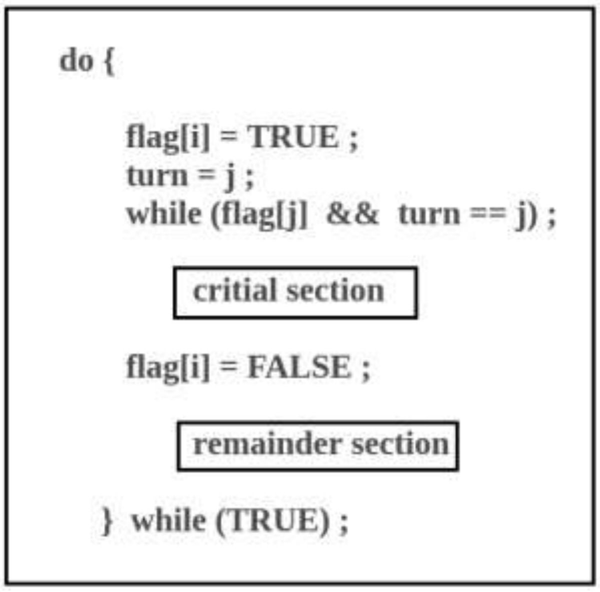
- **Mutual Exclusion** : If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress** : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes 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.

Software Solution to Critical Section Problem

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.



```
do {

      flag[i] = TRUE ;
      turn = j ;
      while (flag[j]  &&  turn == j) ;

          critial section

      flag[i] = FALSE ;

          remainder section

   }  while (TRUE) ;
```

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.

Hardware Solution to Critical Section Problem

Process Synchronization problems occur when two processes running concurrently share the same data or same variable. The value of that variable may not be updated

correctly before its being used by a second process. Such a condition is known as Race Around Condition. There are a software as well as hardware solutions to this problem. In this article, we will talk about the most efficient hardware solution to process synchronization problems and its implementation.
There are two algorithms in the hardware approach of solving Process Synchronization problem:

1. Test and Set
2. Swap

Hardware instructions in many operating systems help in the effective solution of critical section problems.

Hardware synchronization algorithms like Test-and-Set and Swap are essential for preventing race conditions and ensuring proper resource allocation. If you're preparing for GATE and want to understand these algorithms in detail, the GATE CS Self-Paced Course provides a thorough exploration of these synchronization techniques.

1. Test and Set:

Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured.

Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

Test and Set Pseudocode

```
//Shared variable lock initialized to false
boolean lock;

boolean TestAndSet (boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}

while(1){
    while (TestAndSet(lock));
critical section
    lock = false;
remainder section
}
```

2. Swap:

Swap algorithm is a lot like the TestAndSet algorithm. Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section.

Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process). Again on next iteration while(key) is true so this will keep on executing and another process will not be able to enter in critical section. Therefore Mutual exclusion is ensured. Again, out of the critical section, lock is changed to false, so any process finding it gets t enter the critical section. Progress is ensured. However, again bounded waiting is not ensured for the very same reason.

Swap Pseudocode

```
// Shared variable lock initialized to false
// and individual key initialized to false;
boolean lock;
Individual key;

void swap(boolean &a, boolean &b) {
   boolean temp = a;
   a = b;
   b = temp;
}

while (1){
   key = true;
   while(key)
       swap(lock,key);
critical section
   lock = false;
remainder section
}
```

Semaphore

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.

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

Wait

The wait operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed.

```
wait(S)
{
  while (S<=0);
  S--;
}
```

Signal

The signal operation increments the value of its argument S.

```
signal(S)
{
  S++;
}
```

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.

Mutual Exclusion

During concurrent execution of processes, processes need to enter the critical section (or the section of the program shared across processes) at times for execution. It might happen that because of the execution of multiple processes at once, the values stored in the critical section become inconsistent. In other words, the values depend on the sequence of execution of instructions – also known as a race condition. The primary task of process synchronization is to get rid of race conditions while executing the critical section.

What is Mutual Exclusion?

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.

Examples of such resources include files, I/O devices such as printers, and shared data structures.

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.

Approaches To Implementing Mutual Exclusion

- **Software Method:** Leave the responsibility to the processes themselves. These methods are usually highly error-prone and carry high overheads.
- **Hardware Method:** Special-purpose machine instructions are used for accessing shared resources. This method is faster but cannot provide a complete solution. Hardware solutions cannot give guarantee the absence of deadlock and starvation.
- **Programming Language Method:** Provide support through the operating system or through the programming language.

Requirements of Mutual Exclusion

- At any time, only one process is allowed to enter its critical section.
- The solution is implemented purely in software on a machine.
- A process remains inside its critical section for a bounded time only.
- No assumption can be made about the relative speeds of asynchronous concurrent processes.
- A process cannot prevent any other process from entering into a critical section.
- A process must not be indefinitely postponed from entering its critical section.

What is a Need of Mutual Exclusion?

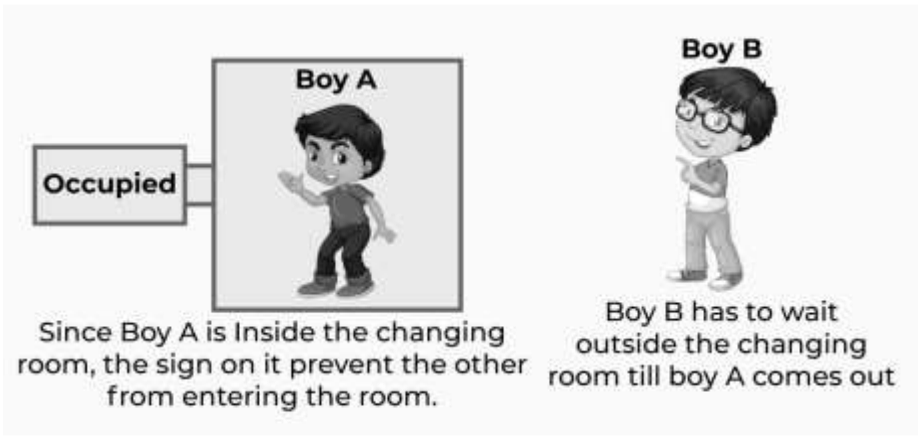
An easy way to visualize the significance of mutual exclusion is to imagine a linked list of several items, with the fourth and fifth items needing to be removed. By changing the previous node's next reference to point to the succeeding node, the node that lies between the other two nodes is deleted.

To put it simply, whenever node "i" wants to be removed, node "with – 1″'s subsequent reference is changed to point to node "ith + 1" at that time. Two distinct nodes can be removed by two threads at the same time when a shared linked list is being used by many threads. This occurs when the first thread modifies node "ith – 1" next reference, pointing towards the node "ith + 1," and the second thread modifies node "ith" next reference, pointing towards the node "ith + 2." Although both nodes have been removed, the linked list's required state has not yet been reached because node "i + 1" still exists in the list because node "ith – 1" next reference still points to it.

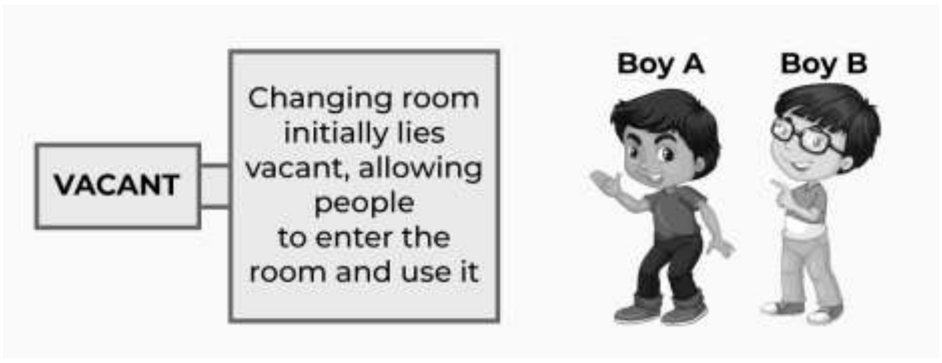
Now, this situation is called a race condition. Race conditions can be prevented by mutual exclusion so that updates at the same time cannot happen to the very bit about the list.

Example:

In the clothes section of a supermarket, two people are shopping for clothes.

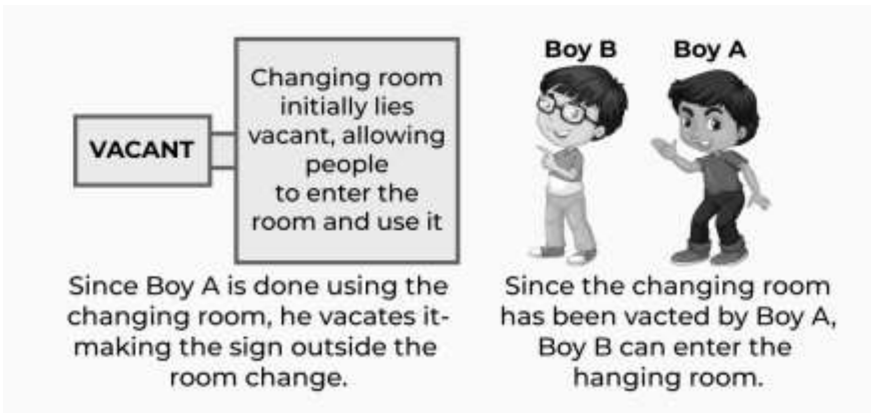

Occupied
Boy A
Boy B
Since Boy A is Inside the changing room, the sign on it prevent the other from entering the room.
Boy B has to wait outside the changing room till boy A comes out

Boy, A decides upon some clothes to buy and heads to the changing room to try them out. Now, while boy A is inside the changing room, there is an 'occupied' sign on it – indicating that no one else can come in. Boy B has to use the changing room too, so she has to wait till boy A is done using the changing room.


VACANT
Changing room initially lies vacant, allowing people to enter the room and use it
Boy A
Boy B

Once boy A comes out of the changing room, the sign on it changes from 'occupied' to 'vacant' – indicating that another person can use it. Hence, boy B proceeds to use the changing room, while the sign displays 'occupied' again.

The changing room is nothing but the critical section, boy A and boy B are two different processes, while the sign outside the changing room indicates the process synchronization mechanism being used.


VACANT
Changing room initially lies vacant, allowing people to enter the room and use it
Boy B
Boy A
Since Boy A is done using the changing room, he vacates it- making the sign outside the room change.
Since the changing room has been vacted by Boy A, Boy B can enter the hanging room.

<u>Conclusion</u>

This week we learned about the Process Synchronization, Race Conditions, Critical Section Problem, Software Solution to Critical Section Problem, Hardware Solution to Critical Section Problem, Semaphore and Mutual Exclusion