Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

Independent Process
Cooperative Process:

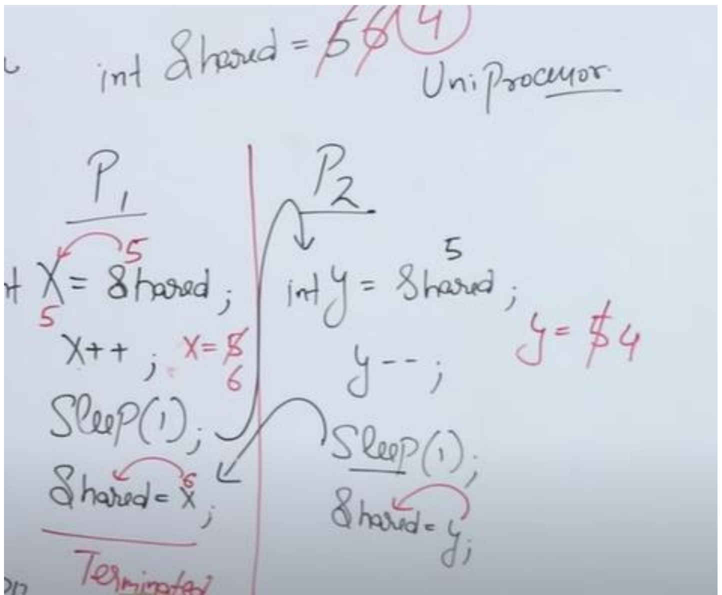
Independent Processes: one process does not affect the execution of another process.

Cooperative Processes: 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.

In the **cooperative proces**s , same data and resource is being shared there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource. Thus, all the processes race to say that my result is correct. This condition is called the **Race Condition. (**CRITICAL SECTION SOLVE THE RACE CONDITION **PROBLEM)**

Or:

When there is more than one process accessing or modifying a shared resource at the same time, then the value of that resource will be determined by the last process. This is called the **race condition.**



Process Synchronization

Phenomenon in which the execution of processes in such a way that no two processes can have access to the same shared data and resources.

It mainly use in the cooperative process

Producer-Consumer problem

- The Producer-Consumer problem is a classic problem this is used for multi-process synchronization

  In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming. he producers and consumers share the same memory buffer that is of fixed-size.

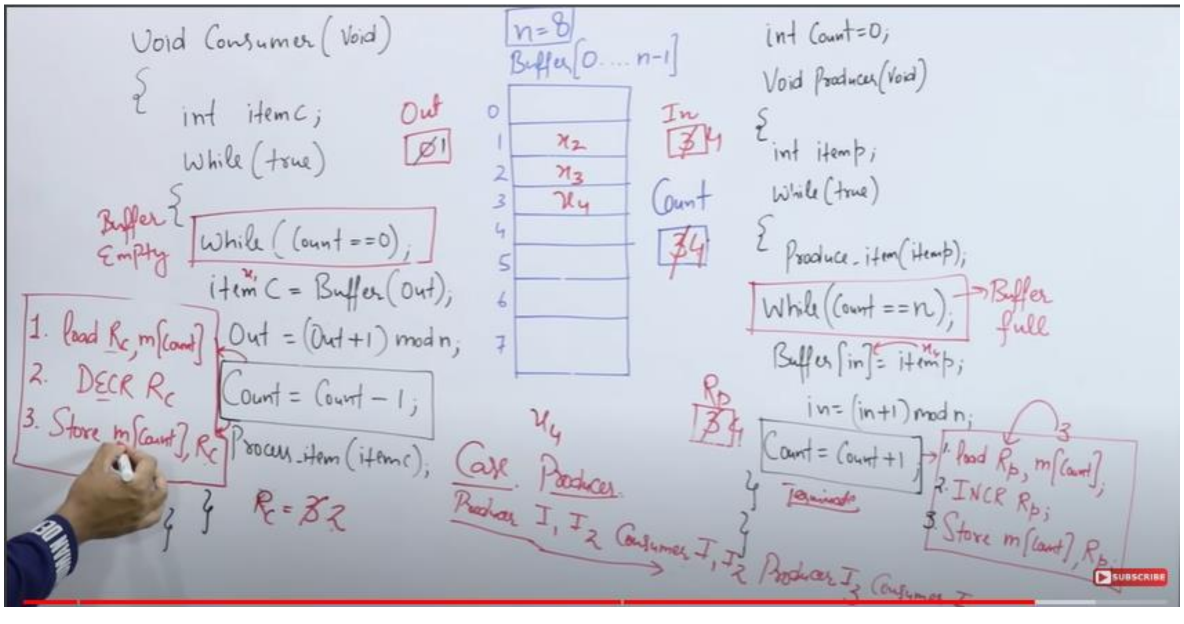
  Job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

  ### Problem:

  The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.

The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.

The producer and consumer should not access the buffer at the same time.



```
Void Consumer(Void)
{
    int itemc;
    While(true)
    {
        While((count==0);
        itemC = Buffer(out);
        Out = (Out+1) mod n;
        Count = Count - 1;
        Process_item(itemc);
    }
}
```

```
1. load Rc, m[Count]
2. DECR Rc
3. Store m[Count], Rc
```

```
int Count=0;
Void Producer(Void)
{
    int itemp;
    While(true)
    {
        Produce_item(itemp);
        While((Count==n);
        Buffer[in]= itemp;
        in= (in+1) mod n;
        Count = Count+1
    }
}
```

```
1. load Rp, m[Count];
2. INCR Rp;
3. Store m[Count], Rp;
```

Reader Writer Problem:

There are multiple processes and only one printer for printing purposes as the printer is a slower device so the spooler comes into the picture. The spooler is a program that maintains a directory containing a list of files that the printer has to print. Spooler hands over the file to the printer one by one.

Execution of a program in a spooler:

I1. LOAD Ri M[IN] {**IN** is a shared variable between processes that want to use the printer. It points to the position where the file can be inserted into the spooler }

I2. STORE SD[Ri] { **Ri** is the register of process Pi}

I3. INCR Ri { **SD** is a spooler directory }

I4. STORE M[IN] Ri

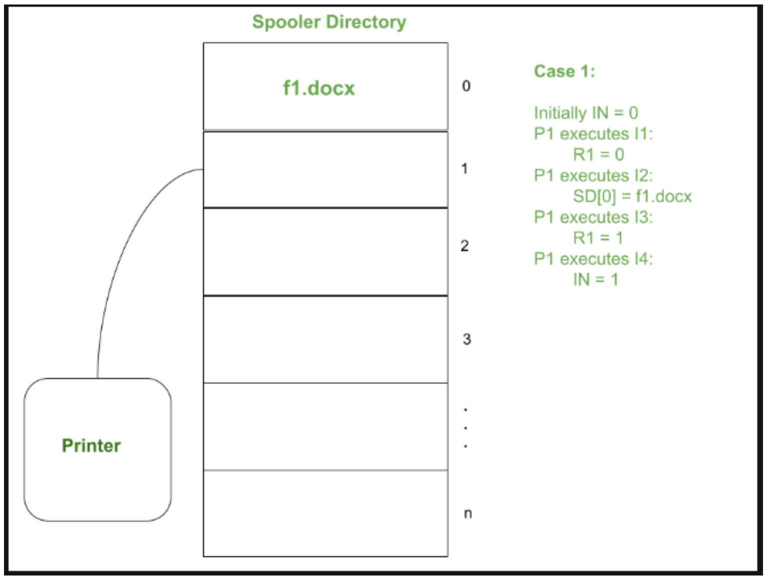
There are two cases:

Only one process wants to use the printer.

More than one processes want to use the printer.

Case 1: (Single Process):

Process P1 wants to print the file "f1.docx", so it starts executing the spooler program:



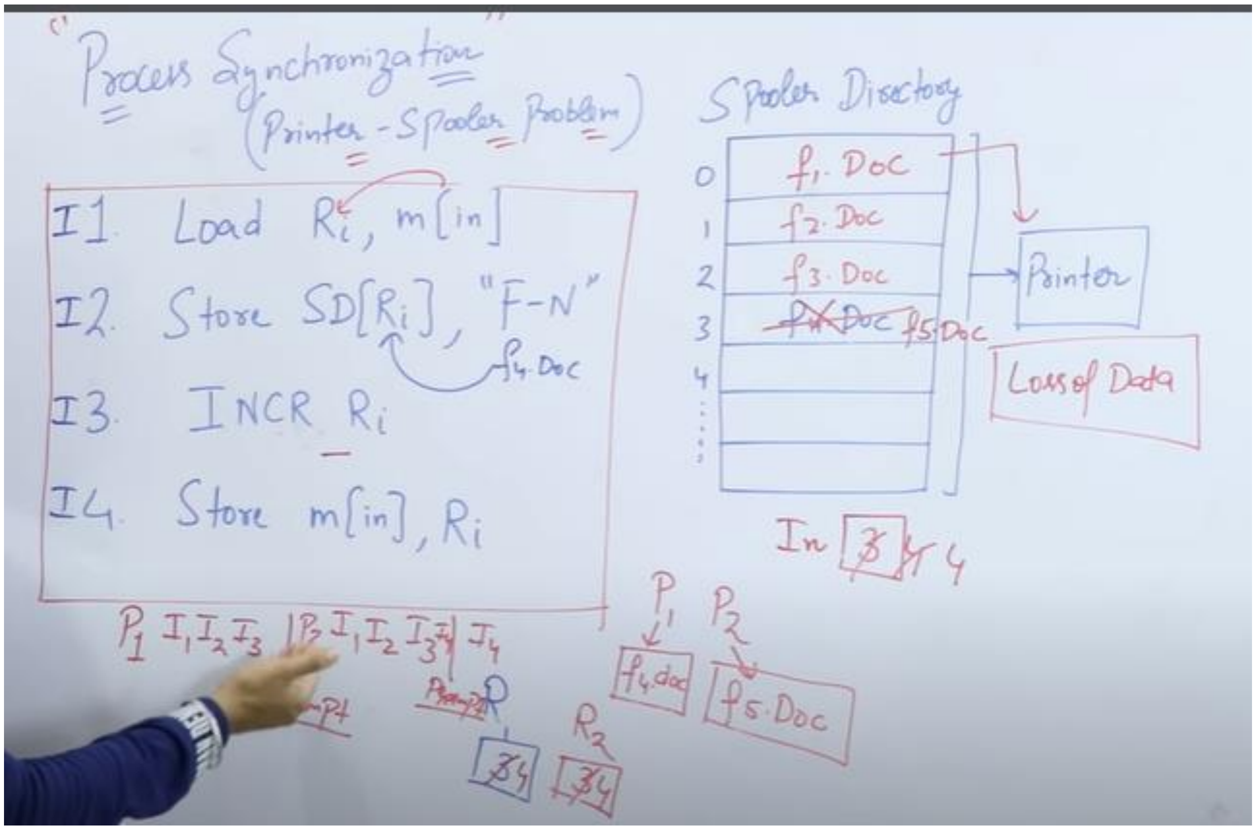
Single Process into execution

Here the process P1 has executed the spooler program successfully, and the file F1 is queued in the spooler directory for printing.

Case 2: (Multiple processes):

Say some files are already present inside the spooler and the value of IN is initially 3.

Here what happened is when P1 executes it stored f4.docx at 3 and when process P2 arrived it also stored f5.docx at the same location by overwriting f4.docx. Hence we suffered a **loss of data**. This loss of data happened due to non-synchronization between P1 and P2 since both processes were sharing the same spooler program.



Critical Section Problem:

Critical section is a code segment / part of the program where the shared variables can be accessed by the various processes {Cooperative}.It's a place where shared resources are placed

Only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

EXAMPLE: WHERE CRITICAL SECTION IS USED(RACE CONDITION)

Consider an account having a balance of ₹10,000. Let us consider that, when a cashier withdraws the money, it takes 2 seconds for the balance to be updated in the account.

It is possible to withdraw ₹7000 from the cashier and within the balance update time of 2 seconds, also withdraw an amount of ₹6000 from the ATM.

Thus, the total money withdrawn becomes greater than the balance of the bank account

Solution to the Critical Section Problems

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**
  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
- **Progress**
  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting**
  Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

Lock Variable use to solve the Critical Section Problem:

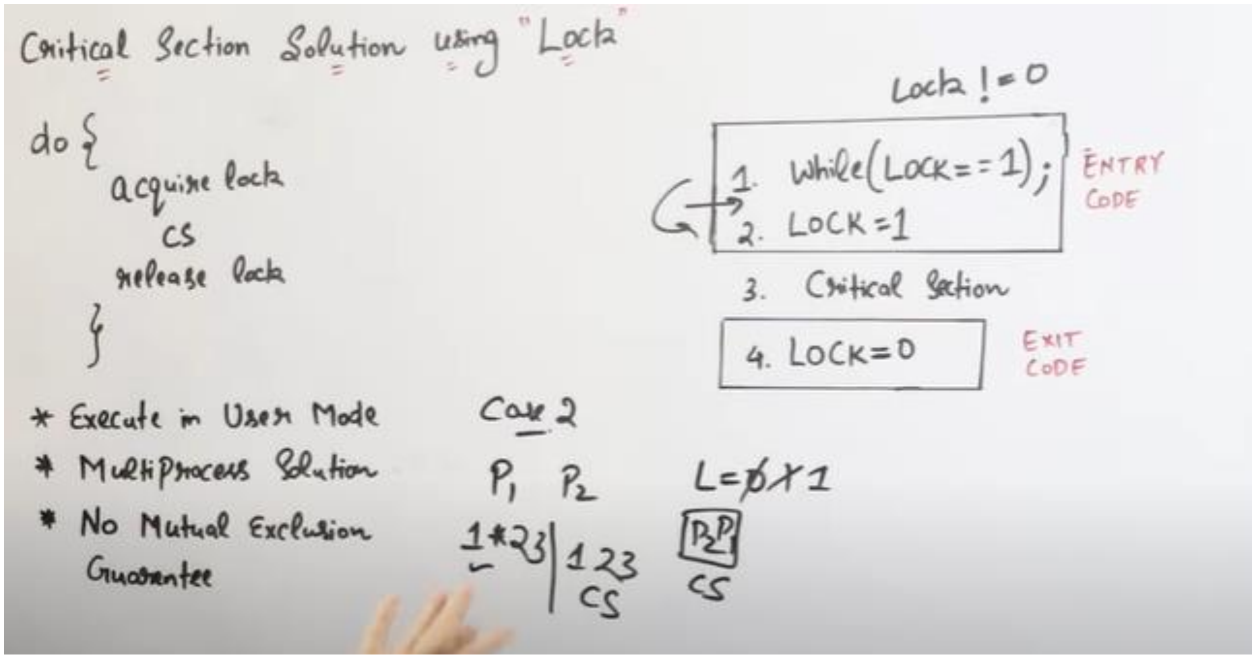
Its a software mechanism implemented in user mode, i.e. no support required from the Operating System.

- Its a busy waiting solution (keeps the CPU busy even when its technically waiting).
- It can be used for more than two processes.
- When Lock = 0 implies critical section is vacant (initial value) and Lock = 1 implies critical section occupied.

- ### Conditions fulfill:
  Mutual Exclusion: no

- The pseudocode looks something like this

```
Entry section - while(lock != 0);
                Lock = 1;
//critical section
Exit section - Lock = 0;
```



Case 1:
WHEN there are two process p1 & p2 : p1 go for the execution so in this way the value of the lock will become 1 and then the p! will enter in the critical section hence it will not stop until the process is not executed and after the execution the lock value will become 0 so in this way other process can go for the execution

Case 2:
when there are two process p1 & p2: and p1 execute the statement 1 and get pre-empted
then P2 executes statement 1 and enter into the critical section and in this way P1 come back again and we know it already execute the statement 1 so it will enter in the critical section as well

According to the condition of mutual exclusion, more than one process in the critical section must not be present at the same time. Hence, the lock variable mechanism doesn't guarantee the mutual exclusion

EXAMPLE:
Classic implementation of the reader-writer's problem. The buffer here is the shared memory and many processes are either trying to read or write a character to it. To prevent any ambiguity of data we restrict concurrent access by using a lock variable

Solution: For mutual exclusive in lock variable is test and set instruction:

Test and Set Instruction:

Test and Set Lock (TSL) is a synchronization mechanism.

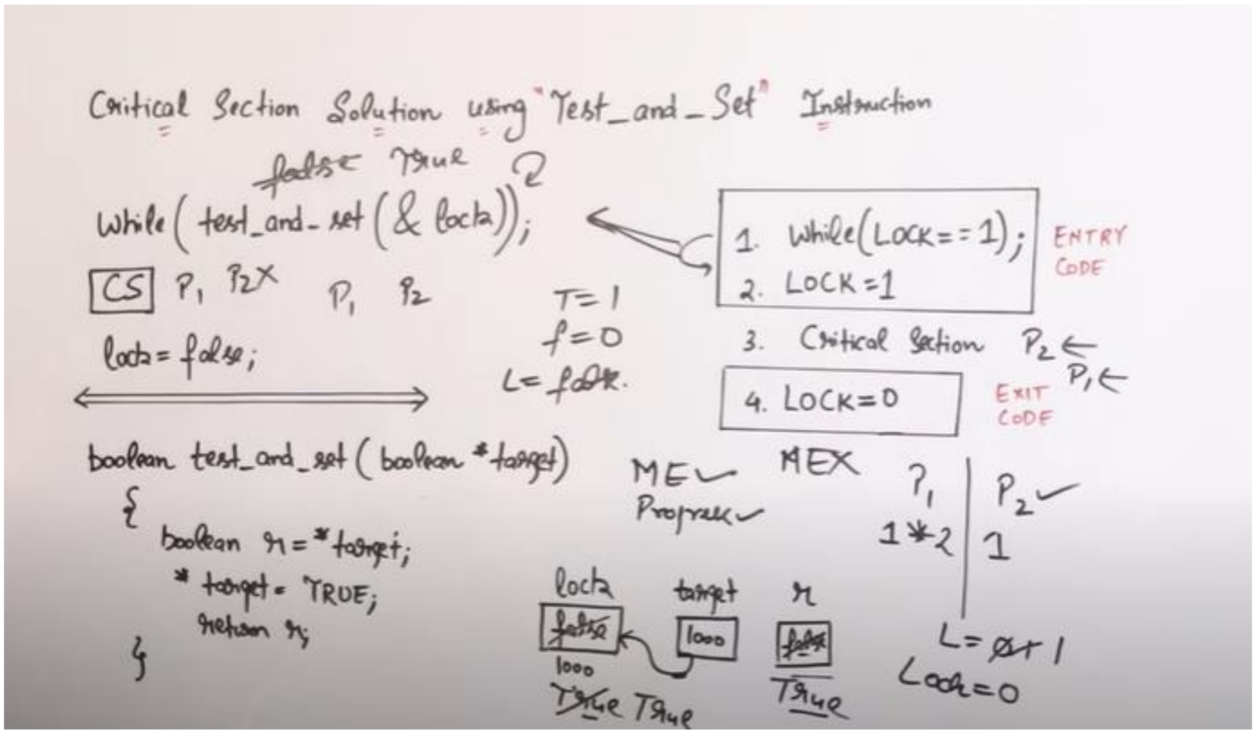
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently

Conditions fulfill:

Mutual Exclusion: yes
Progress: yes
Bounded Waiting: yes

Case 1:
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.

Case 2:
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.



Turn Variable:

Turn variable is a synchronization mechanism that provides synchronization among two processes.

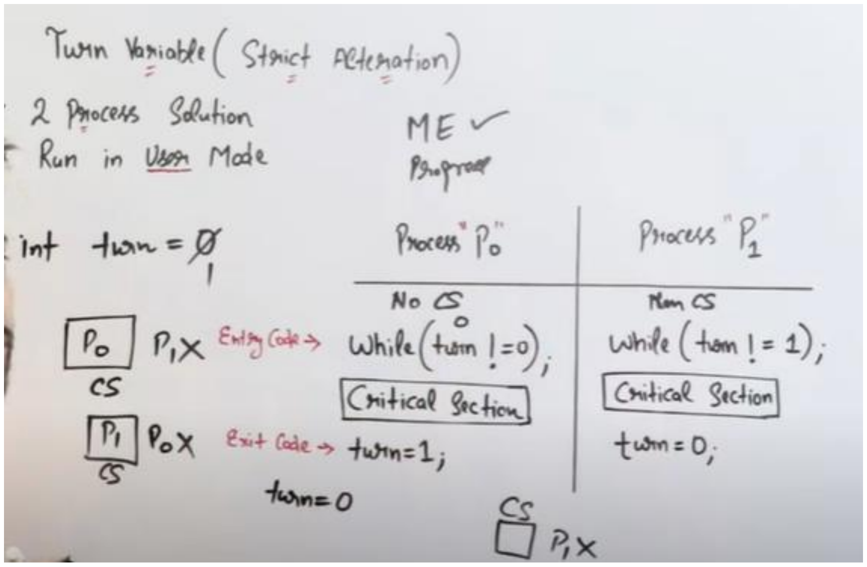
It is implemented on user mode and trict alteration approach on two processes

Conditions fulfill:

Mutual Exclusion: yes

Progress: no

Bounded Waiting: yes



Turn Variable (Strict Alternation)
2 Process Solution
Run in User Mode
ME✓
Progress
int turn = 0 1
Process "P0"
Process "P1"
No CS
Non CS
P0 P1X
CS
Entry Code →
while (turn != 0);
while (turn != 1);
Critical Section
Critical Section
P1 P0X
CS
Exit Code →
turn=1;
turn = 0;
turn=0
CS
P1X