

CRITICAL SECTION(CS)

There are two type of processes:

⇒ Co-operative : execution of one process affects the execution of other process. these process share some resources.

⇒ Independent: execution of one process doesn't affect another.

race around condition : if two process access the same part or shared resources at same time.

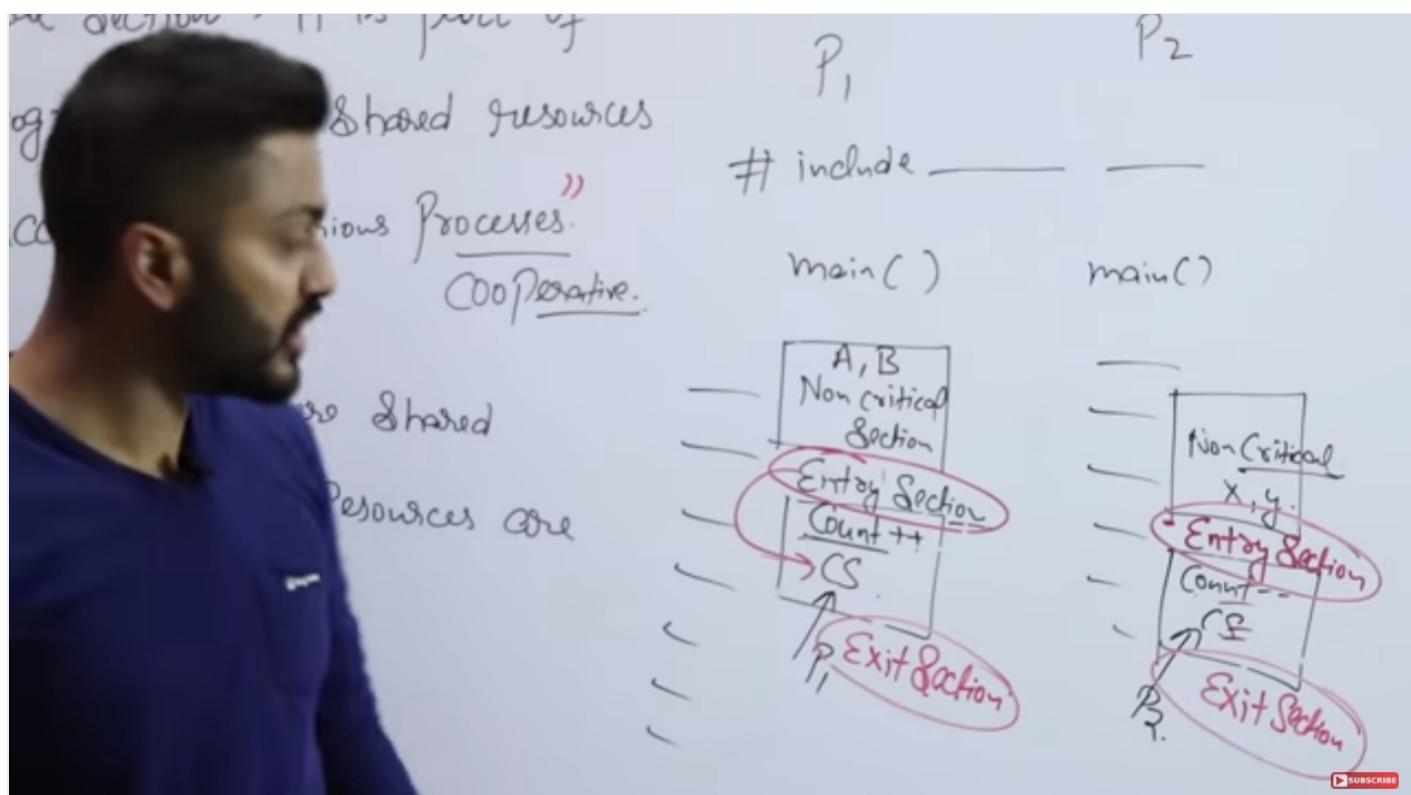
→condition arises when the process is not done in the proper sequence.

if two processes are accessing the same part and the output depend on the order of execuition, the condition is called race around condition.

to avoid the condition we have to avoid it.

critical section:

It is the part of the program where shared resources are accessed by various processes (Co-operative processes)



we have to execute some code present in the entry section before entering the critical section and we have to execute some code before exiting the critical section.

To achieve synchronization there can be several methods/solution but these 4 conditions should be satisfied:

1. Mutual exclusion
2. progress
3. bounded wait
4. no assumption related to hardware speed

→ 1 and 2 are primary rules, 3 and 4 are secondary rules.

MUTUAL EXCLUSION :

If there is a process in the critical section then no other process can enter the critical section.

PROGRESS :

If a process wants to enter in critical state but other process is stopping it (there might be some code of process 2) which is blocking the entrance of process 1 in CS, then there is no progress

BOUNDARY WAIT :

There should be a certain number of times a process can repeatedly access the critical state.

It should not be like process1 has used the CS infinite time whereas process2 has used the CS only once. It should not be the case.

After a certain number of times process2 should also use the CS.

NO ASSUMPTION RELATED TO HARDWARE AND SPEED :

The solution should not be limited by the hardware and the speed of system.

Like : your solution can run only on LINUX or it can only work on 64-bit systems.

Critical section solutions :-

1. LOCK variable :

→ Acquire lock before going to CS

→ Access the CS

→ Release the lock

Case 1

Critical Section Solution using "Lock"

```
do {
    acquire lock
    CS
    release lock
}
```

- * Execute in User Mode
- * Multiprocess Solution
- * No Mutual Exclusion Guarantee

$Lock_1 = 0$

1. While($LOCK == 1$);
2. $LOCK = 1$

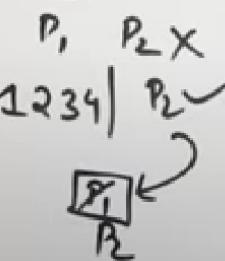
ENTRY
CODE

3. Critical Section

4. $LOCK = 0$

EXIT
CODE

Case 1



$Lock = 0 \rightarrow \text{Vacant}$

$1 \rightarrow \text{Free}$

$Lock = \emptyset \times \emptyset$

case 2 if p1 execute line 1 and preempt or stops due to some reason.

Critical Section Solution using "Lock"

```
do {
    acquire lock
    CS
    release lock
}
```

- * Execute in User Mode
- * Multiprocess Solution
- * No Mutual Exclusion Guarantee

$Lock_1 = 0$

1. While($LOCK == 1$);
2. $LOCK = 1$

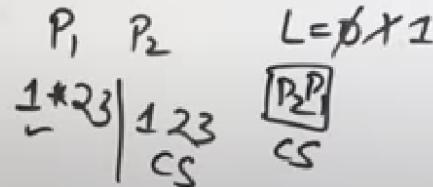
ENTRY
CODE

3. Critical Section

4. $LOCK = 0$

EXIT
CODE

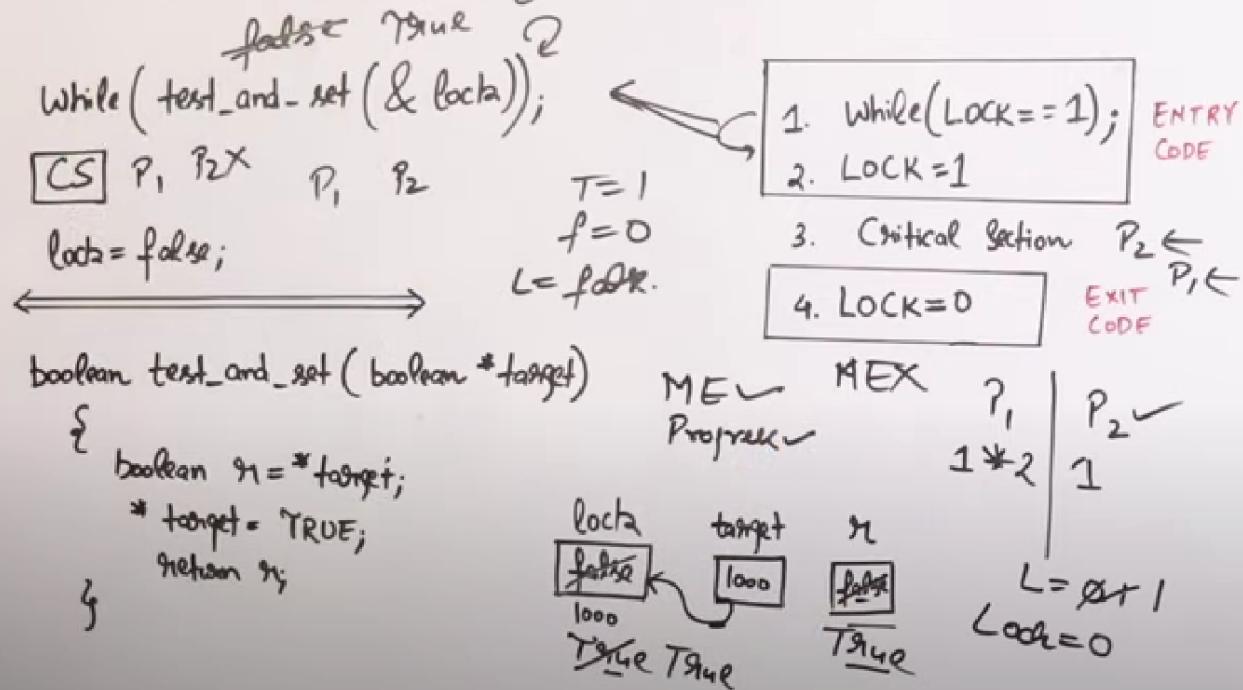
Case 2



mutual exclusion was not satisfied by case 2

2. Test and set :

Critical Section Solution using "Test_and_Set" Instruction



mutual exclusion and progress is satisfied.

3. TURN VARIABLE :

Process2 can only access critical section if process1 have already accessed it otherwise it can't be accessed. hence satisfying the condition of mutual exclusion but not the progress, as without process1 process2 can't access CS.

it is also called strict alteration after accessing CS once by P1, it can't again access it until P2 doesn't access it.

Turn Variable (Strict Alteration)

- * 2 Process Solution
- * Run in User Mode

int turn = 0;

P₀

P_1, X Entry Code \rightarrow

CS

P₁

P_0, X Exit Code \rightarrow

Process "P₀"

No CS

`While (turn != 0);`

Critical Section

`turn = 1;`

Process "P₁"

Non CS

`While (turn != 1);`

Critical Section

`turn = 0;`

Twin Variable (Strict Alternation)

Process Solution
in User Mode

ME ✓
Progress X

BW ✓

twin = 0



P₀X

CS



P₁X

CS

Process "P₀"

No CS
while (twin != 0);

Critical Section

Entry Code → twin = 1;

twin ->

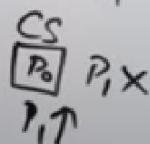
Process "P₁"

No CS

while (twin != 1);

Critical Section

twin = 0;



P₀X

P₁X

P₁T

4. PETERSON'S SOLUTION

Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j .

Peterson's solution requires two data items to be shared between the two processes:

int turn

→ Indicates whose turn it is to enter its critical section.

boolean flag [2]

→ Used to indicate if a process is ready to enter its critical section.

NESO ACADEMY



① Peterson's algorithm - (for 2 processes)

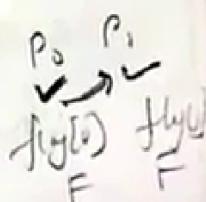
int turn;

"I need it, but first you have, if you also need"

boolean flag[2]; /* initially set to false */

P₀:

do



```
flag[0]=true; /* intend to enter critical section */
turn=1;
while ( flag[1] && turn==1); /* keep looping as long as
                           flag[1] is set to true &
                           turn=1 */
                           /* entry section */
<critical section>
```

```
flag[0]=false; /* exiting from critical section */
```

```
<remainder section>
} while(1);
```

P₁:

```
do
  flag[1]=
```

```
<remainder section>
} while(1);
```

P₁:

```
do
  flag[1]=true;
  turn=0;
```

```
while ( flag[0] && turn==0);

```

```
<critical section>
```

```
flag[1]=false;
```

```
<remainder section>
```

```
} while(1);
```

Mutual exclusion is satisfied.

Progress is satisfied

Bounded waiting is satisfied.

PUSH PULL MIGRATION :

In multiprocessors, whichever process have more processes to process will push their process to the processor who is having few number of process. same the processor with less number of processes will pull the processes.

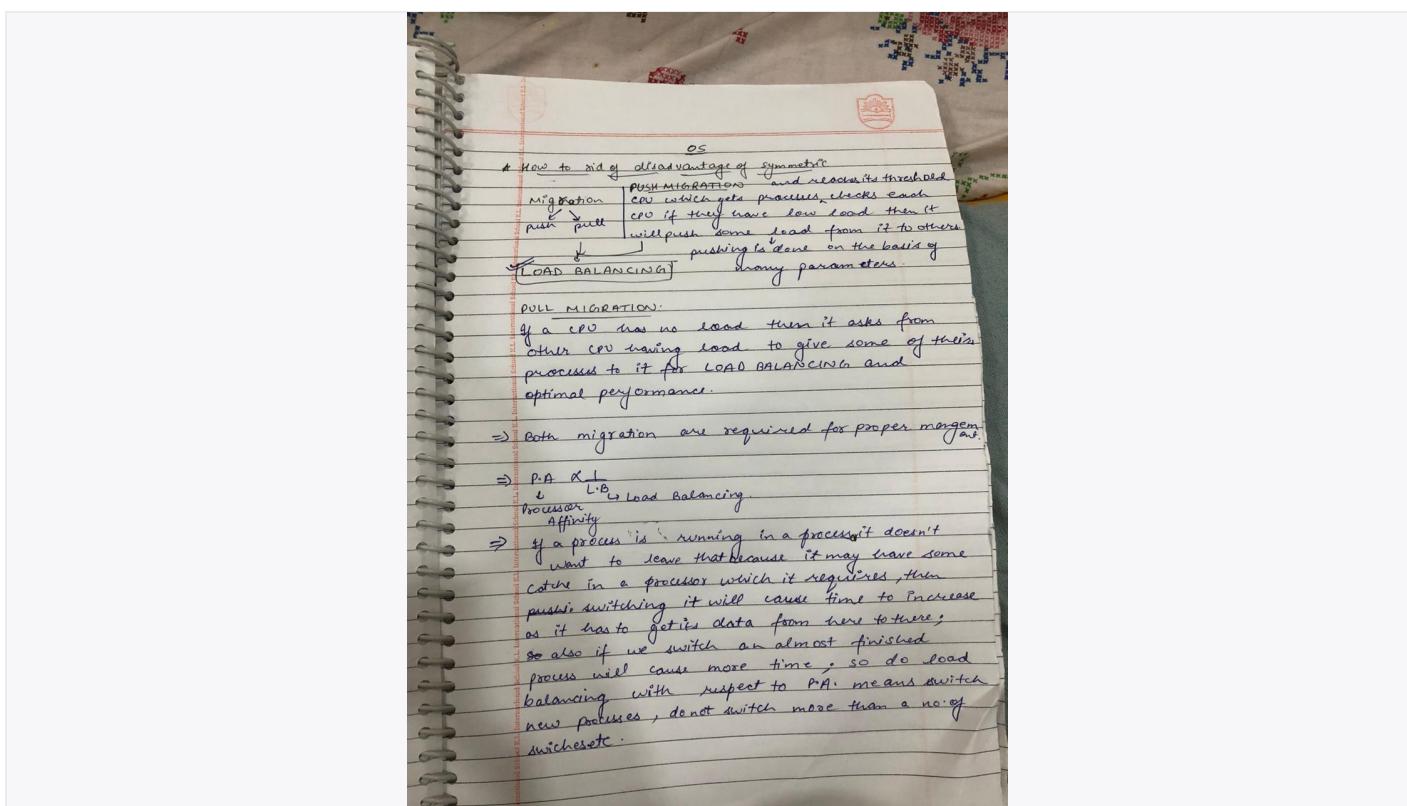
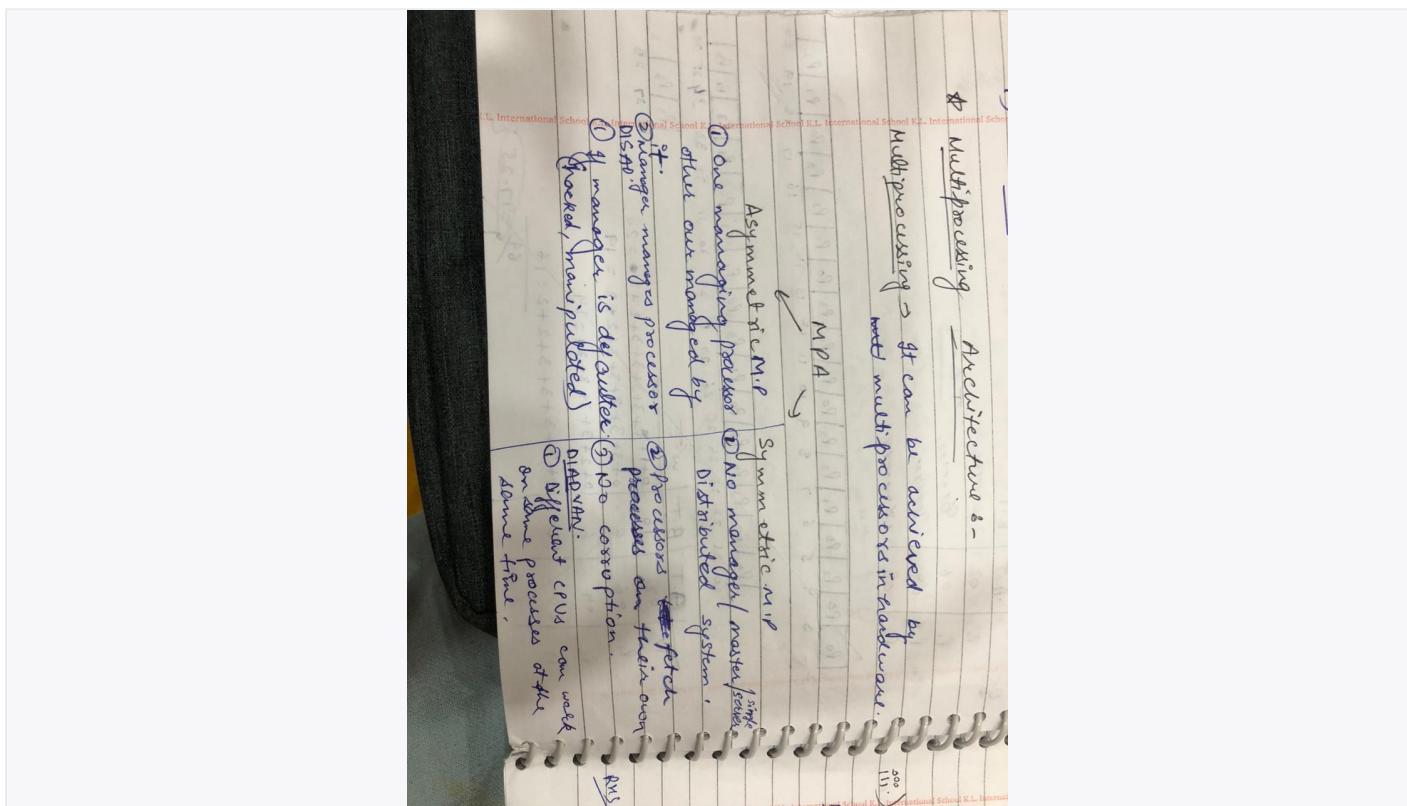
Multiprocessor architecture: multiple processors in the hardware.

the system which have more than 2 processors.

they are of 2 types:

symmetric

assymmetric multiprocessing



SHORTEST REMAINING TIME FIRST IS THE PREEMPTIVE VERSION OF SJF.

Advantages:

SRTF algorithm makes the processing of the jobs faster than SJN algorithm, given it's overhead charges are not counted.

Disadvantages:

The context switch is done a lot more times in SRTF than in SJN, and consumes CPU's valuable time for processing. This adds up to it's processing time and diminishes it's advantage of fast processing.

