Q1. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows: **[GATE 2008]**

Q1. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows: **[GATE 2008]**

```
void enter_CS(X)
{
    while(test-and-set(X));
}

void leave_CS(X)
{
    X = 0;
}
```

Q1. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows: **[GATE 2008]**

```
void enter_CS(X)
{
    while(test-and-set(X));
}

void leave_CS(X)
{
    X = 0;
}
```

In the above solution, **X** is a memory location associated with the **CS** and is initialized to **0** .
Now consider the following statements:

Q1. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows: **[GATE 2008]**

```
void enter_CS(X)
{
    while(test-and-set(X));
}

void leave_CS(X)
{
    X = 0;
}
```

In the above solution, **X** is a memory location associated with the **CS** and is initialized to **0** .
Now consider the following statements:

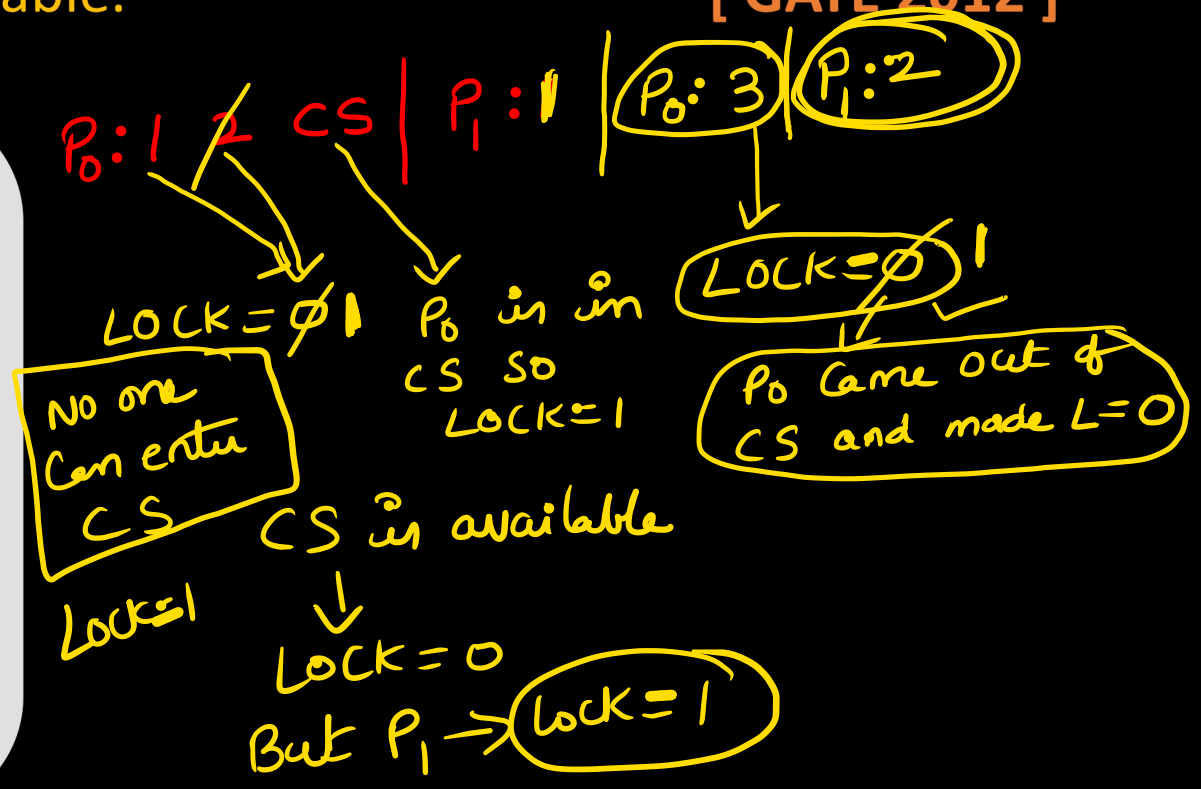
I. The above solution to **CS** problem is deadlock-free
II. The solution is starvation free
III. The processes enter **CS** in FIFO order
IV. More than one process can enter **CS** at the same time

Which of the above statements are TRUE?

(I) only
(I) and (II)
(II) and (III)
(IV) only

Q2. Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available. **[GATE 2012]**

```
AcquireLock(L){
    1) while (Fetch_And_Add(L,1))
    2)     L = 1;
}
ReleaseLock(L){
    3) L = 0;
}
```



Q2. Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available. **[GATE 2012]**

This implementation

(A) fails as L can overflow

(B) fails as L can take on a non-zero value when the lock is actually available

(C) works correctly but may starve some processes

(D) works correctly without starvation

overflow
so it wont work

Lock = 0, but may show Lock != 0.

Q3. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned. **[GATE 2010]**

Q3. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned. [GATE 2010]

```
Method Used by P1
while (S1 == S2) ;
Critical Section
S1 = S2;

Method Used by P2
while (S1 != S2) ;
Critical Section
S2 = not (S1);
```

ME

Bounded waiting

loop: $S_1 != S_2$

loop: $S_1 == S_2$

$P_2\ P_1\ P_2\ P_1\ P_2 \ldots$

$P_2\ P_1\ P_2\ P_1\ P_2 \ldots$

$P_1\ P_2\ P_1\ P_2 \ldots$

P_2 P_1

Progress is not guaranteed.

Strict alternation

	S_1	S_2	
1	T	T	P_2
2	T	F	P_1
3	F	T	P_1
4	F	F	P_2

only one process can enter.

Even if P_1, P_2 read latest values at the same time, still only one can enter
→ ME is guaranteed

Starvation: P_2 → X CS
P_1 → CS

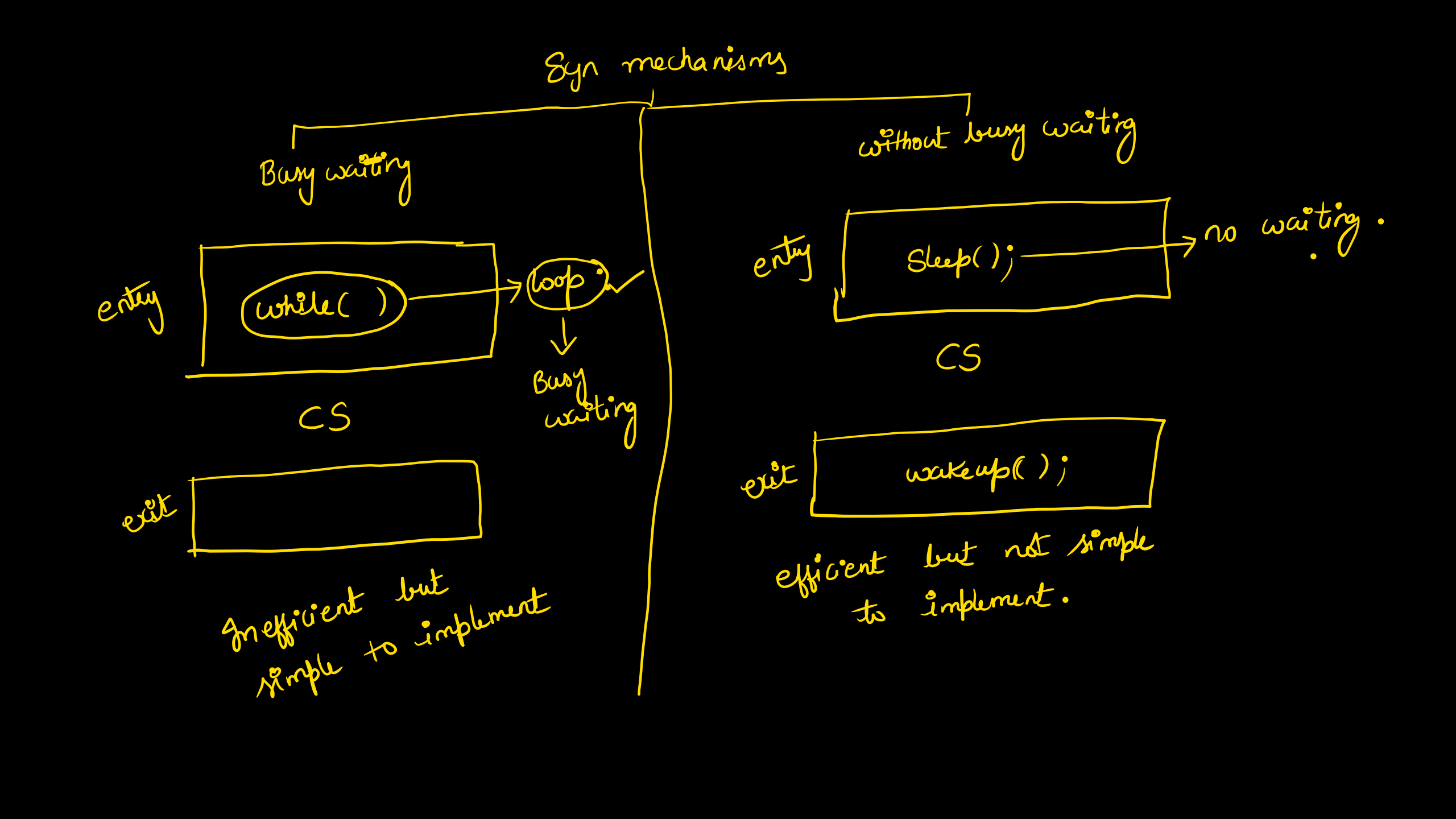
X Progress X

→ (1) →

but P_2 doesn't want CS. P_1 wants CS. P_2 cannot enter

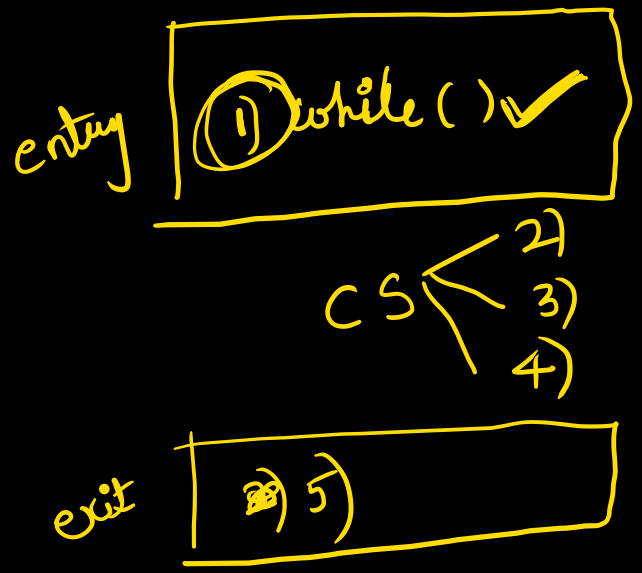
Which one of the following statements describes the properties achieved?

(A) Mutual exclusion but not progress
(B) Progress but not mutual exclusion
(C) Neither mutual exclusion nor progress
(D) Both mutual exclusion and progress


Syn mechanisms
Busy waiting
Without busy waiting
entry
while( )
loop
Busy waiting
CS
exit
Inefficient but simple to implement
entry
Sleep( );
no waiting.
CS
exit
wakeup( );
efficient but not simple to implement.

there is no solution

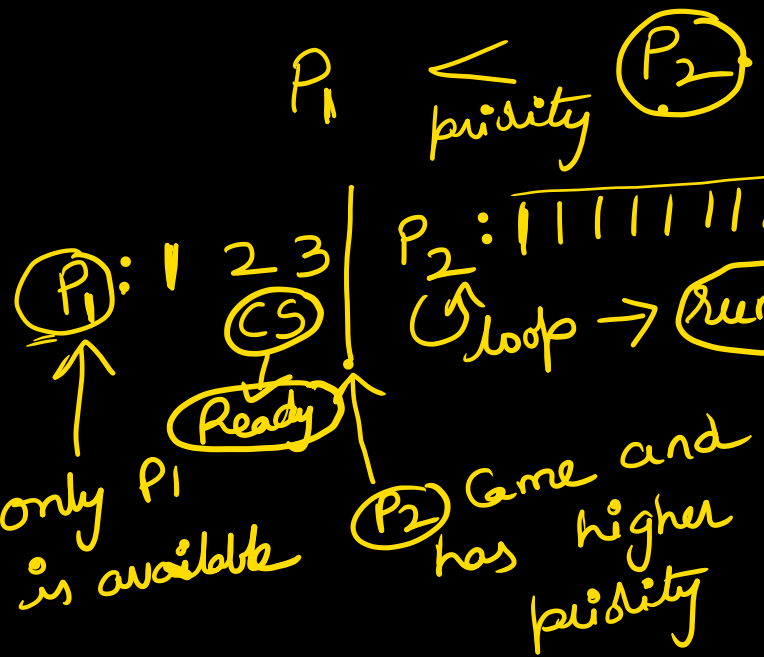
Busy waiting solutions have priority inversion (or) spin lock.



P_1 → ready
P_2 → running

↓

not deadlock

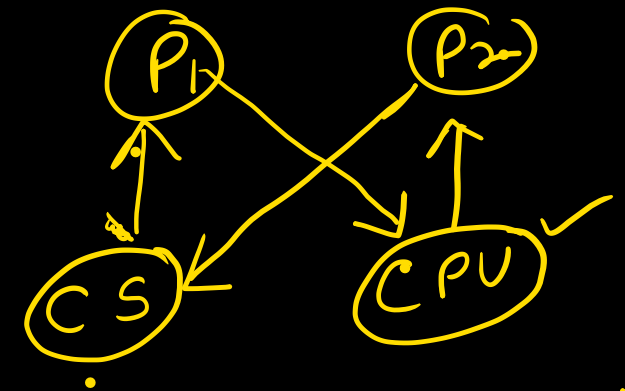


Both will not complete execution

Spin lock of priority inversion not dead lock.

P_1 will never get a chance because P_2 has high priority.

Dead lock in when all the processes are blocked and waiting for each other.



all busy waiting solution have priority inversion problem.

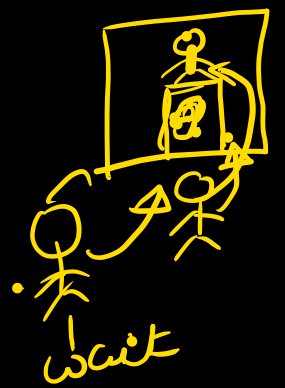
Lock variable:

S/w mechanism implemented in user mode → no support from OS required.

Busy waiting solution (loop)

It works for two or more processes.

Lock = 0 → CS is available
Lock = 1 → CS is not available.



Lock = 0 - not locked
Lock = 1 → locked

In case of Busy waiting solutions

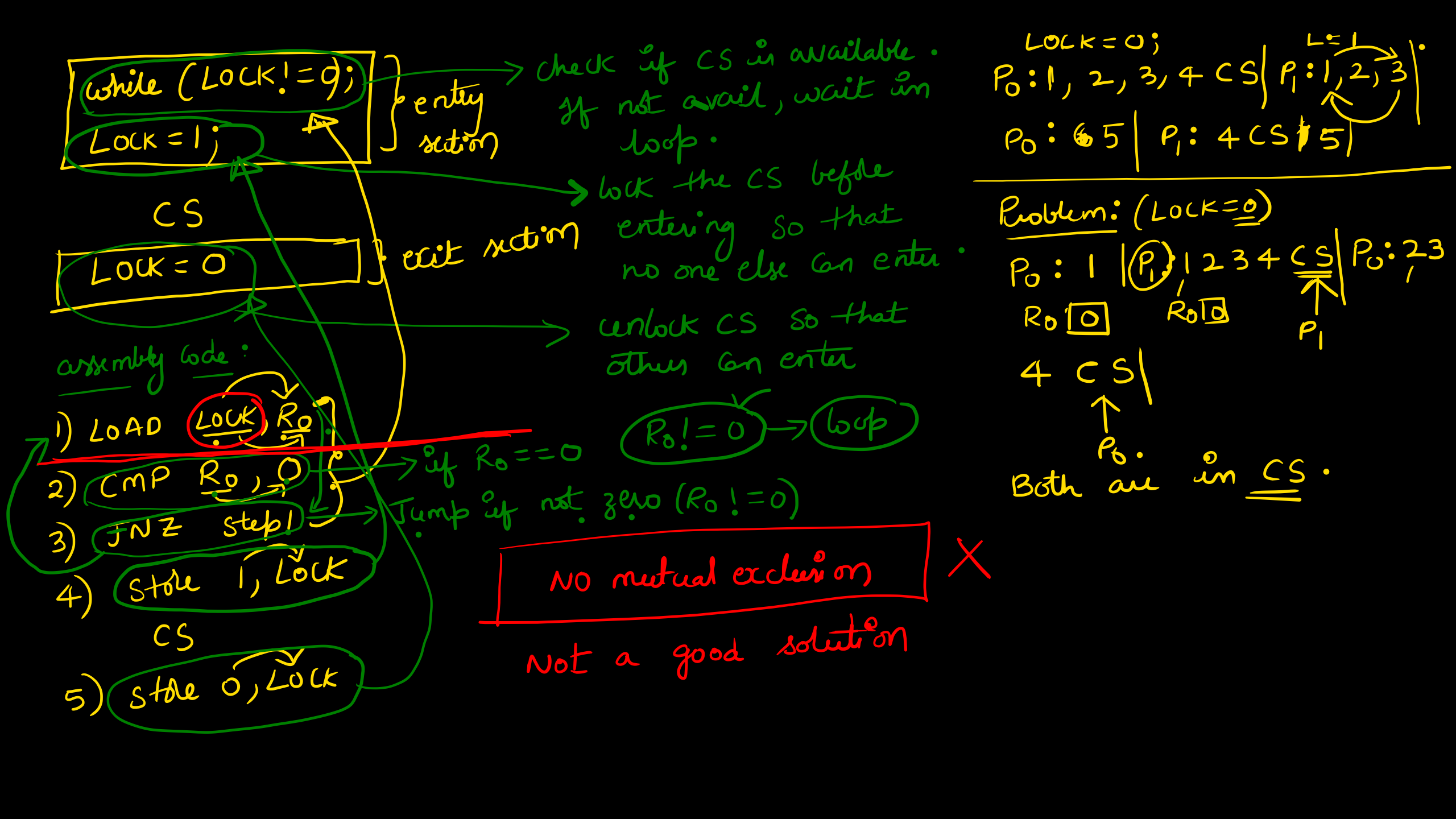
↓

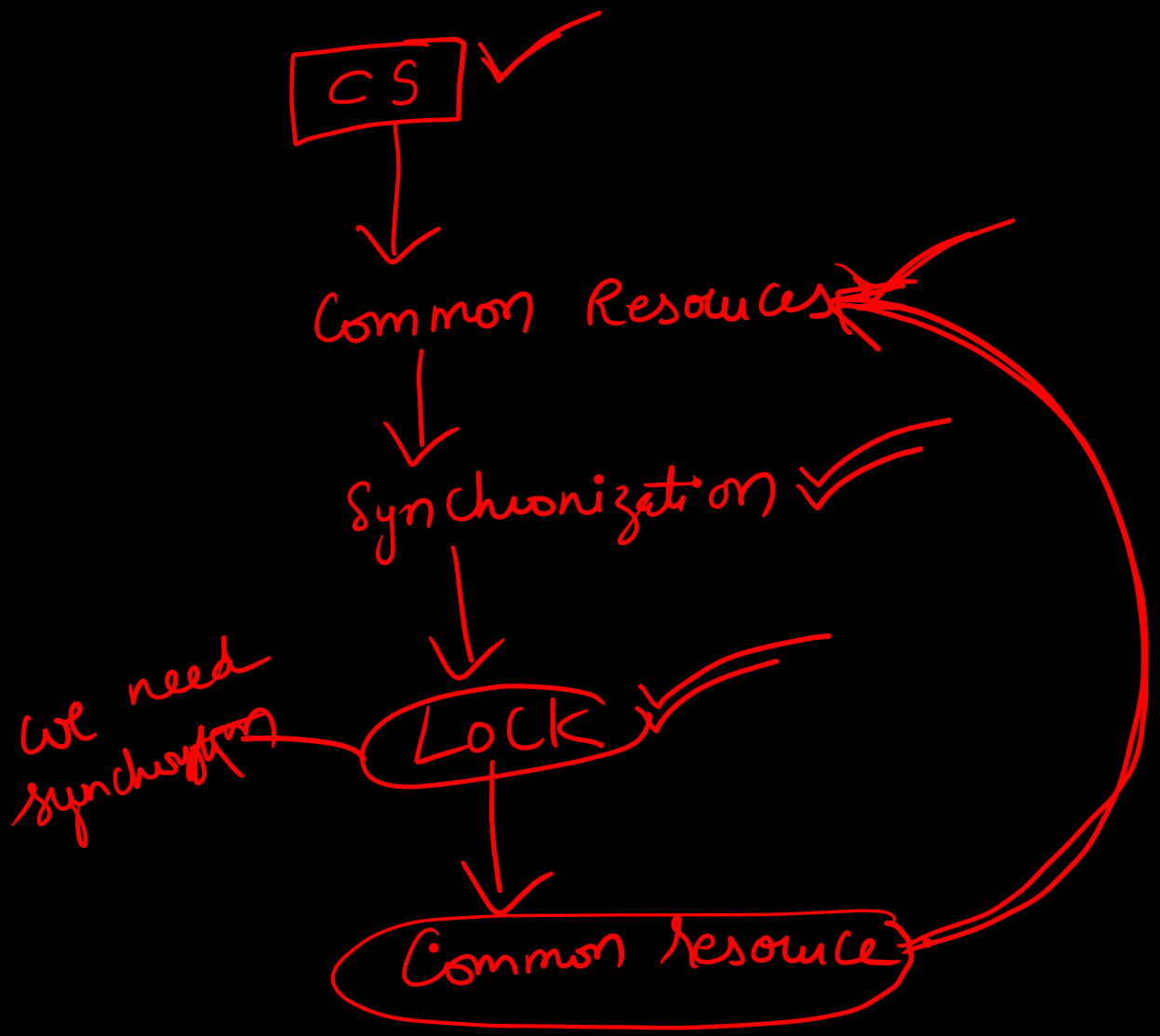
avoid priorities.

If BW + priorities

↓

Priority inversion.


while (LOCK != 0);
LOCK = 1;
entry section
check if CS is available. If not avail, wait in loop.
CS
lock the CS before entering so that no one else can enter.
LOCK = 0
exit section
unlock CS so that others can enter
assembly code:
1) LOAD LOCK, R0
2) CMP R0, 0
3) JNZ step1
4) store 1, LOCK
CS
5) store 0, LOCK
if R0 == 0
Jump if not zero (R0 != 0)
R0 != 0 → loop
NO mutual exclusion
X
Not a good solution
LOCK = 0;
P0: 1, 2, 3, 4 CS | P1: 1, 2, 3
L = 1
P0: 5 | P1: 4 CS 5
Problem: (LOCK = 0)
P0: 1 | P1: 1 2 3 4 CS | P0: 2 3
R0 0
R0 0
P1
4 CS
P0.
Both are in CS.



Rats in house
To kill rats
we got a snake.
now kill the snake.

Lets analyse why we got the problem? LOCK=0

1) load LOCK, R0 → seeing the lock value.
2) CMP R0, 0
3) JNZ step1
4) store 1, LOCK → Locking the CS

Some other process also sees the same LOCK value

P_0: 1 | P_1: 1 | P_2: 1 | P_3: 1 | ....
R0 0 | R0 0 | R0 0 | R0 0

all of them will enter CS at same time.

There is gap between seeing value and Locking it. → we have to close the gap.

P_0: 1 | P_1: 1 | P_2: 1 .... all will enter CS again
R_0 0 | R0 0 | R0 0

.1) load LOCK, R0
.4) store 1, LOCK
.2) CMP R0, 0
.3) JNZ step1

Same meaning

preemption

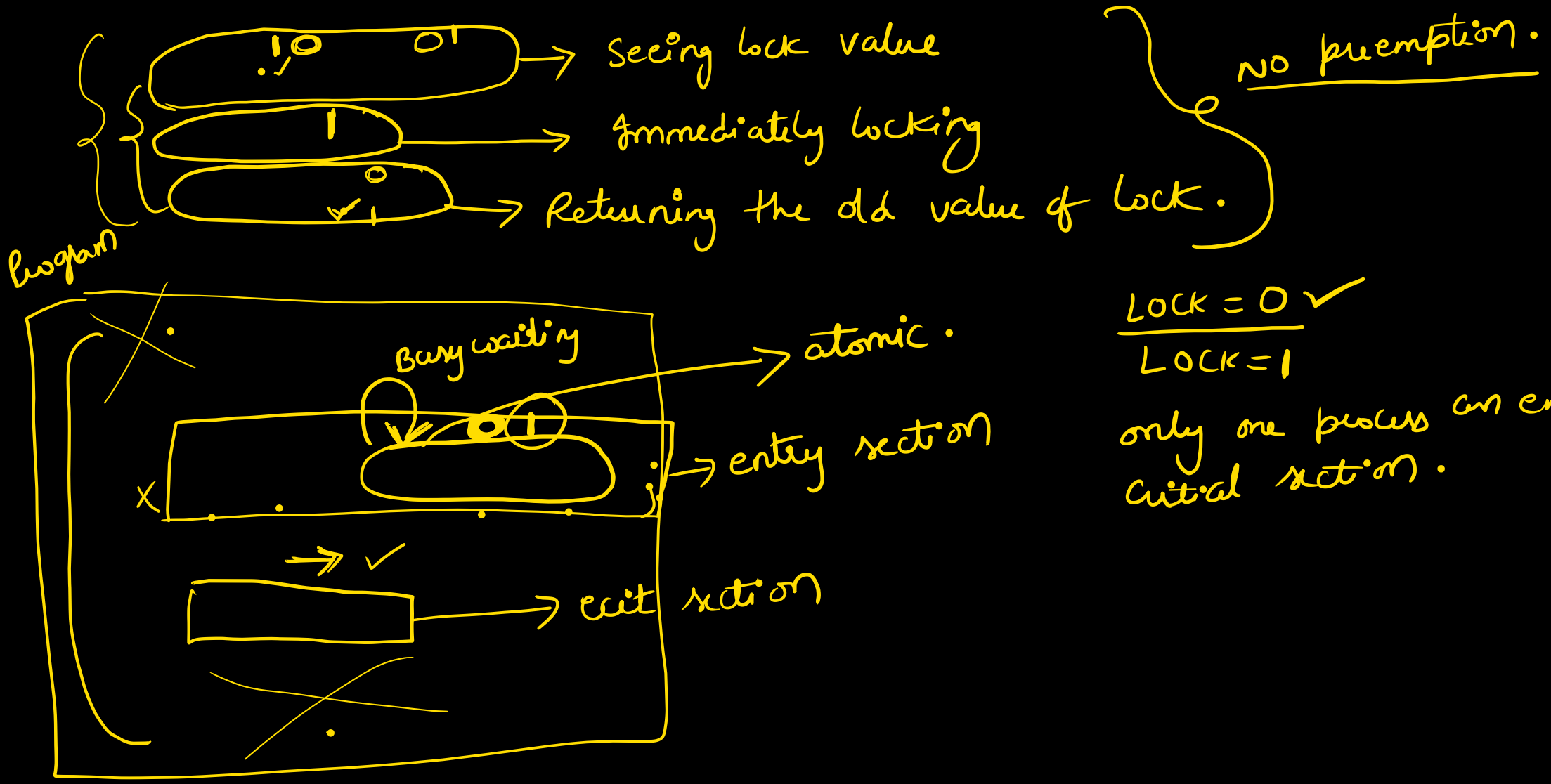
no preemption

But how?

we need support from OS.

→ Atomic → like single instruction

→ TSL → Test set Lock


Seeing lock value
Immediately locking
Returning the old value of lock.
No preemption.
Program
Busy waiting
atomic.
entry section
exit section
X
LOCK = 0 ✓
LOCK = 1
only one process can enter critical section.

4 Conditions: of TSL:

1) mutual exclusion ✓ → only one process will entu CS.
2) Progress → if a process is not interested in CS, it will not stop others
3) ~~Bounded waiting~~ → NO
4) ~~architectural neutral~~ → if need support from H/w and OS

```
do
{
1) while (TSL (LOCK));
2) CS
3) LOCK = 0;
} while (true)
```

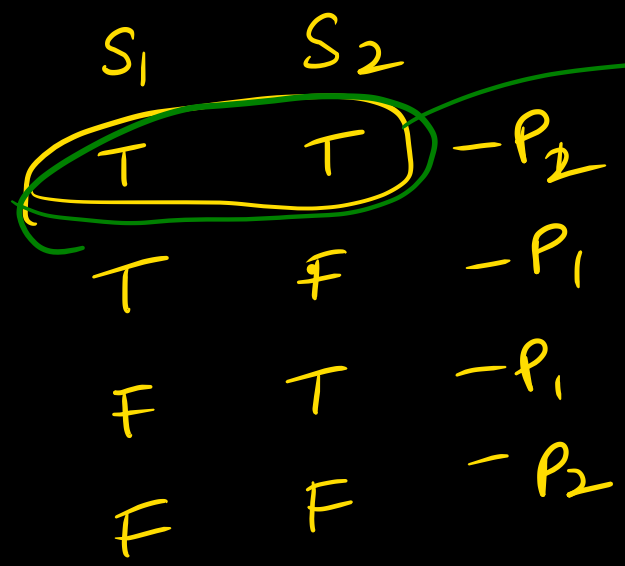
All platforms (OS + H/w) may not have TSL.

P_0 : 1 2 | P_1 : 11111..... | P_0 : 3, 1, 2 |

P_0 in CS — P_1 Busy waiting — P_0 again went to CS

P_0, | 1111..... | P_0 : 3 1 2 | P_1 : 111111..... | - - -..

P_0 again CS — we dont know when P_1 will get CS.



whose chance? → P_2

If But P_2 doesn't want CS.

P_1 wants CS, but it Cannot enter.

only after P_2 enters, P_1 can enter.

why are we getting Race Conditions

1) shared resources
2) Preemption. → what if we eliminate preemption.

↓

caused by interrupts → so disable interrupts.

disable interrupts ←
CS ← only process in CS
enable interrupts ←

Problem: we cannot give control over interrupts to users. They may misuse.