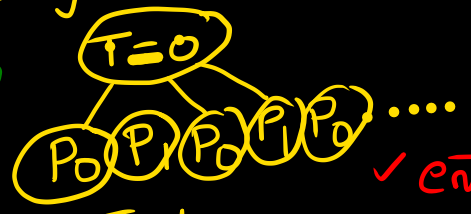
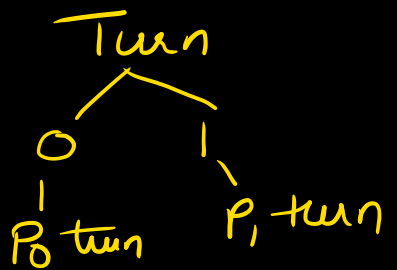
Strict alternation: or Turn variable: (P_0, P_1)

S/W mechanism implemented at user mode. Busy waiting solution.

2 process solution → only 2 processes





For process P_0: → Turn=0

Non CS

while (turn != 0);

CS ✓

turn = 1; ✓ → give chance to P_1

non CS

exit ✓

entry ✓

Mutual exclusion ✓

Progress X

Bounded waiting ✓

Architectural neutrality ✓.

For process P_1: → CS → T=1

non CS

while (turn != 1); ✓ entry

CS ✓

turn = 0; → give chance to P_0

non CS

exit ✓

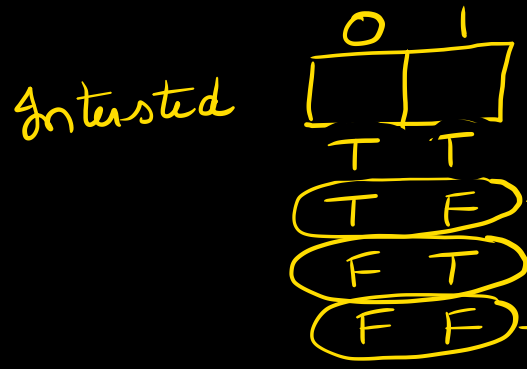
loop.

Progress:

Turn = 0, P_0 is not interested to enter CS.

But P_1 wants CS, still P_1 will be blocked until P_0 enters CS.

Interested variable: S/w mech, 2 process sol, user mode, Busy waiting.



→ mutual ex is guaranteed

P0:

non CS

show the interest

entry

Interested [0] = T;

if P1 is also interested then wait

while (interested [1] == T);

CS

Interested [0] = F → exit

P1:

non CS

show the interest

entry

Interested [1] = T

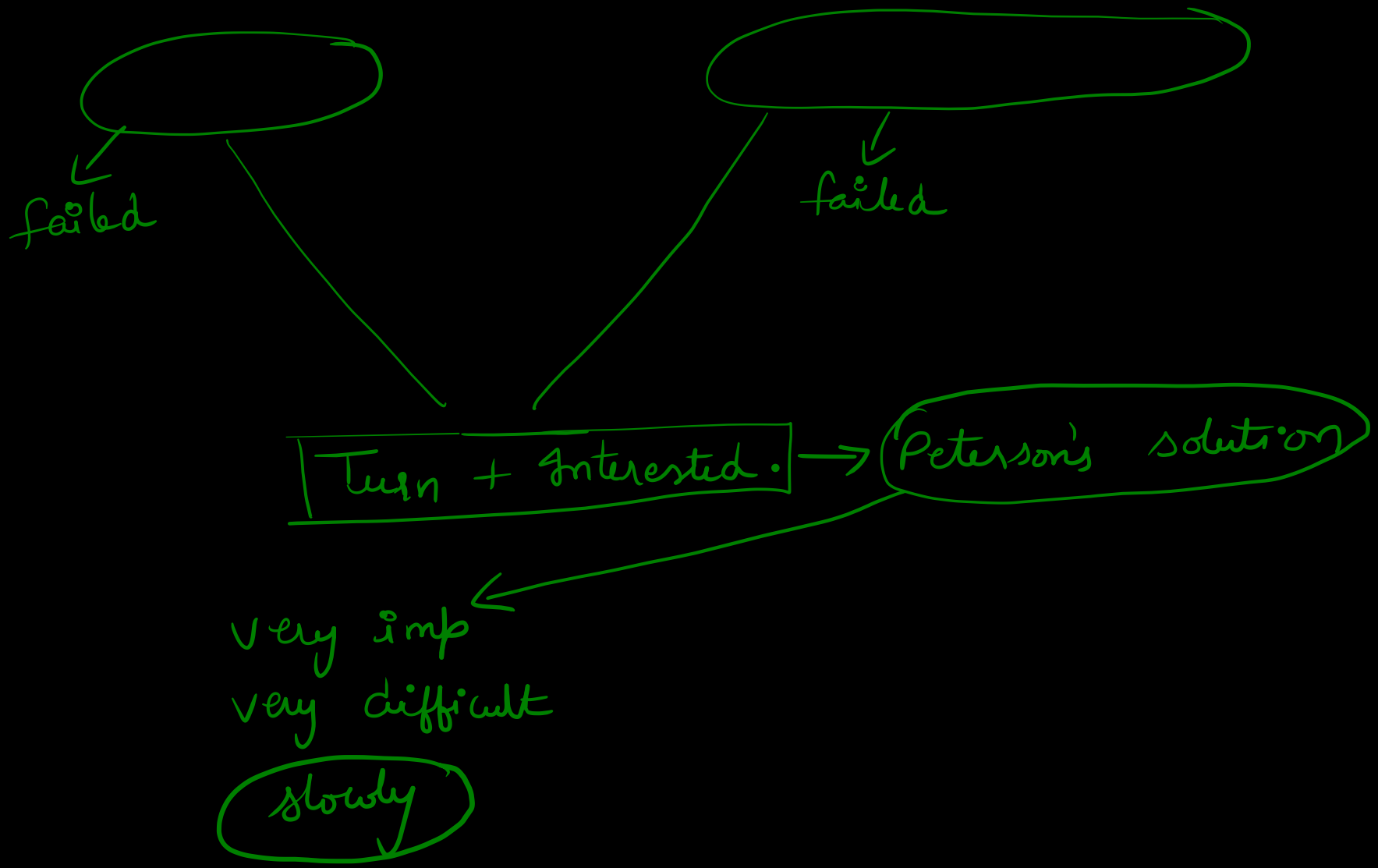
if P0 is also interested then wait

while (interested [0] == T);

CS

exit

Interested [1] = False


failed
failed
Turn + Interested.
Peterson's solution
very imp
very difficult
slowly

Peterson's solution:

→ S/w mechanism at user mode

→ Busy waiting solution

→ (2 process solution) ✓

→ (turn + interested)

P
- mutual exclusion ✓
- Progress ✓
- Bounded wait ✓
- Arch neutrality ✓

→ all four properties satisfied

all these methods are just for practice.
In exam, they will give a new method completely.
So we have to practice a lot of examples
to understand. All these are examples
and we will do many more.
It takes at least 2 minutes to solve a
synchronization question. If you practice well, then
you can solve in 1 min. So practice all the
questions that we do in class and practice dpp's.

diff b/w Starvation / deadlock.

Starvation :

1 or few processes waiting for CS.
At least one process gets CS.

deadlock :

All processes waiting.
No process in getting CS.

```
#define N 2
#define TRUE 1
#define FALSE 0
int interested[N] = FALSE
int turn;
void entry_section (int process)
{
    int other;
    other = 1 - process
    interested[process] = TRUE;
    turn = process
    while (interested[other] == true && turn == process);
}
void exit_section (int process)
{
    interested[process] = False;
}
```

interested [F F] turn [Ø/1/0]

P_0 1 2 3 | P_1: 1 2 3 | P_0: 4 5 | P_1: 4 5 | P_0: 5 CS 6 | P_0: 1 2 3 4 5

P_0: CS | P_1: CS | P_0: CS | P_1: CS. | P_1: 5 CS 6 | P_0: 5 CS

P_1

Bounded waiting.

not sure. Benefit of doubt.

Before entering CS show interest

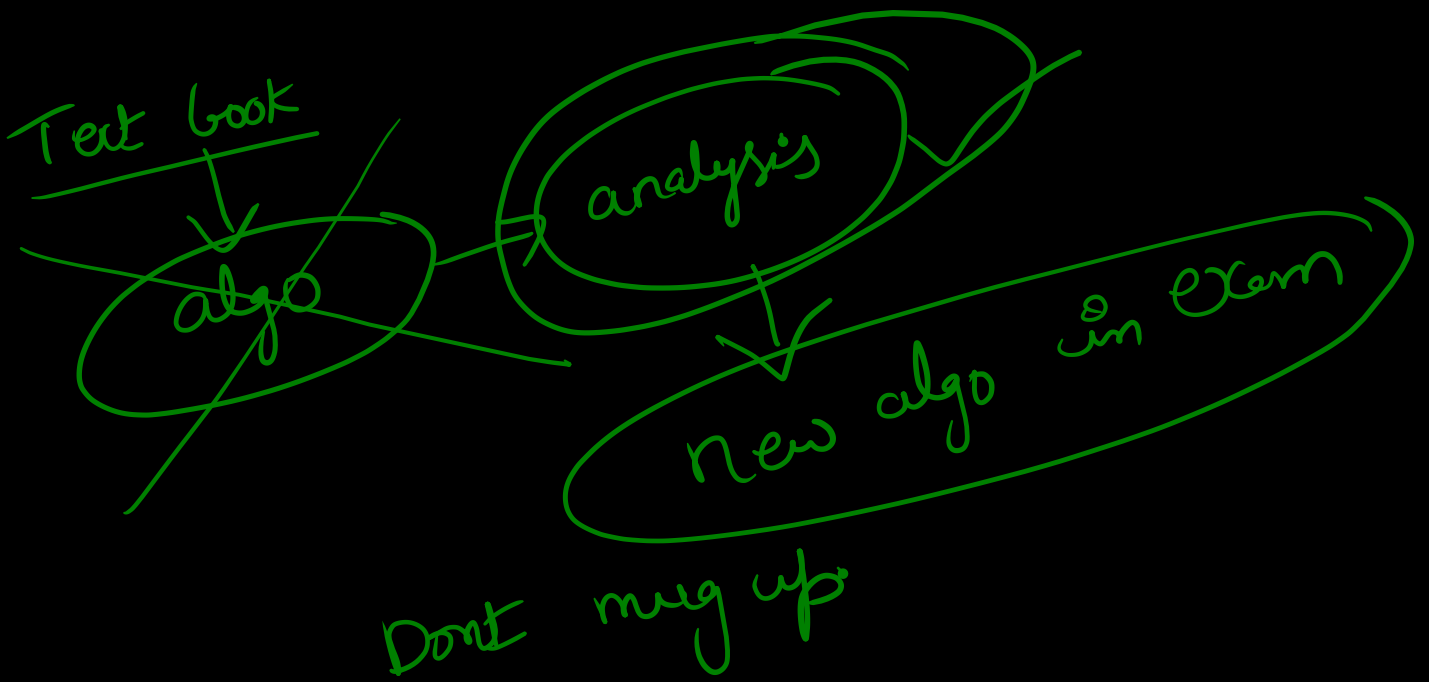
declare that its your turn

if other is interested and its your turn, then wait.

P_0 1 2 3 4 | P_1 1 2 3 4 5 |

Who will enter CS first?

P_0 : 1 2 3 | P_1 : 1 2 3 | P_0 : 4 5 | P_1 : 4 5 | P_0 : 5 CS 6 | P_1 : 5 CS 6 |



Dont mug up

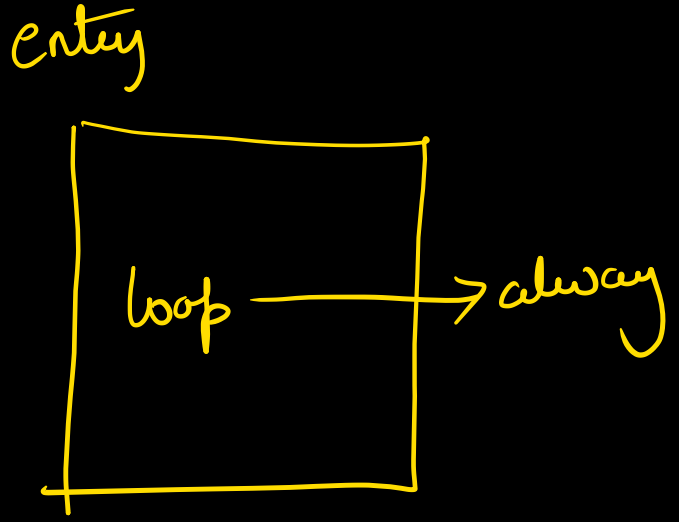
uninterested process, should not stop interested process.

B (W) → waiting.

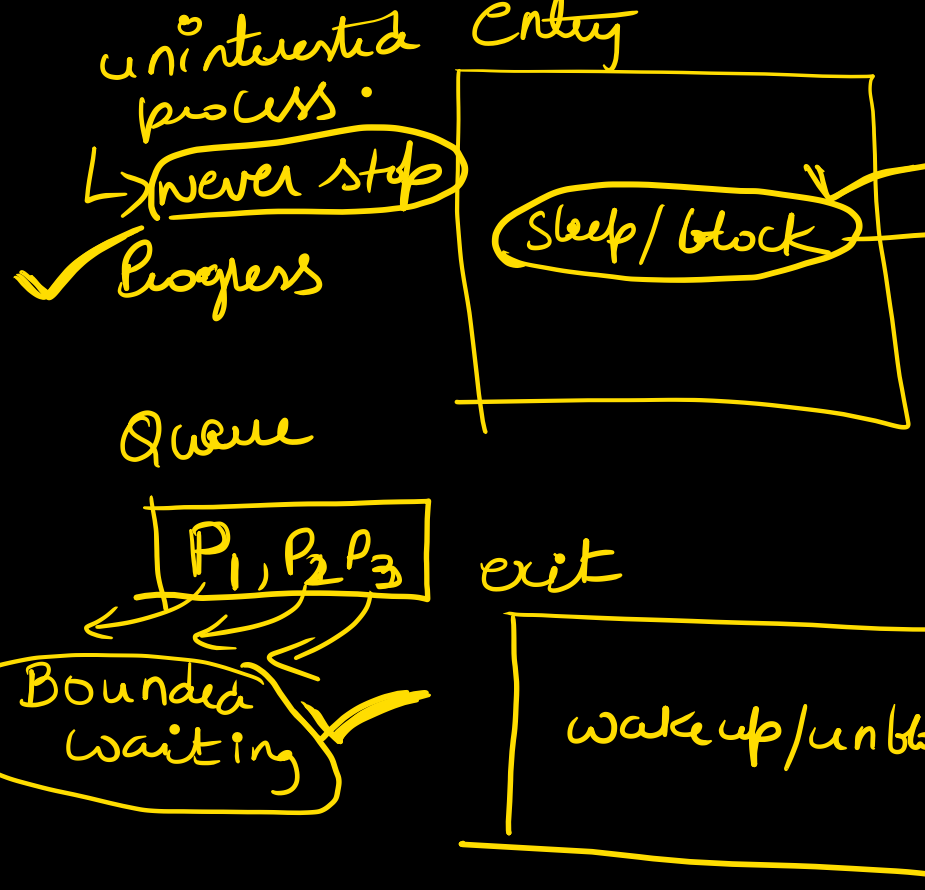
→ A process should not wait forever.

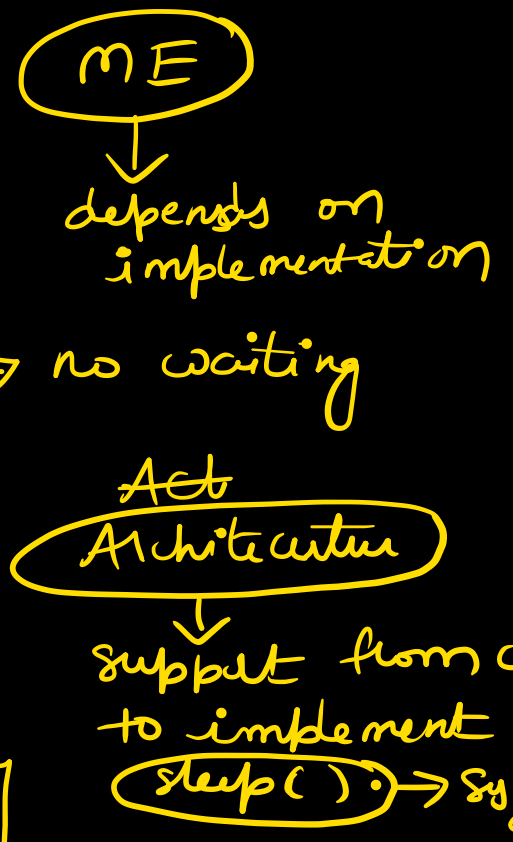
Synchronization mechanisms without Busy waiting

Busy waiting:

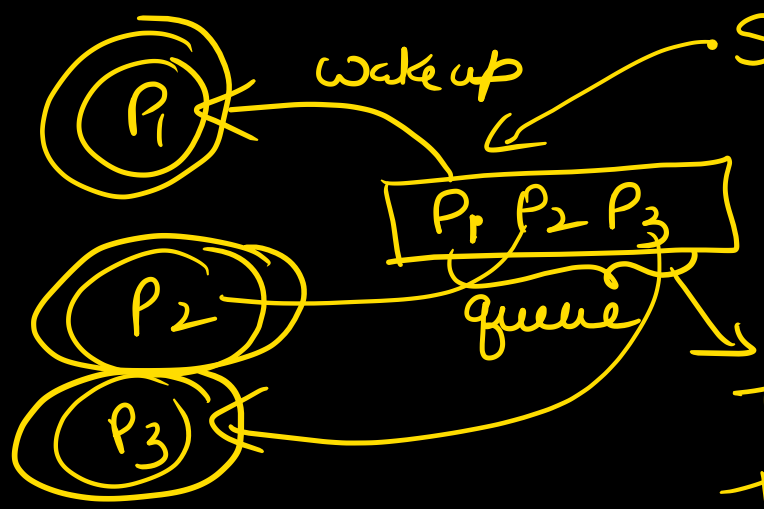


non Busy waiting:



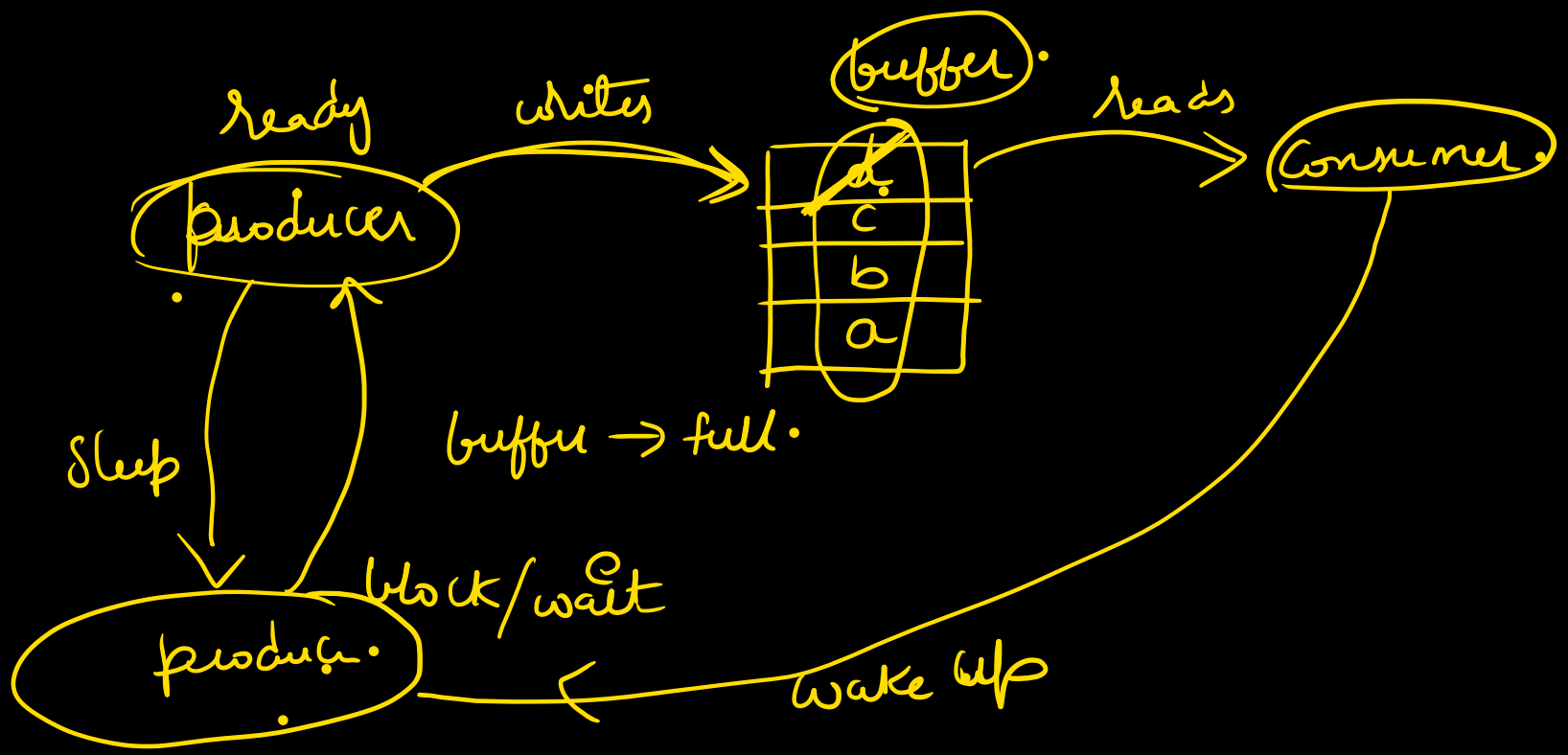


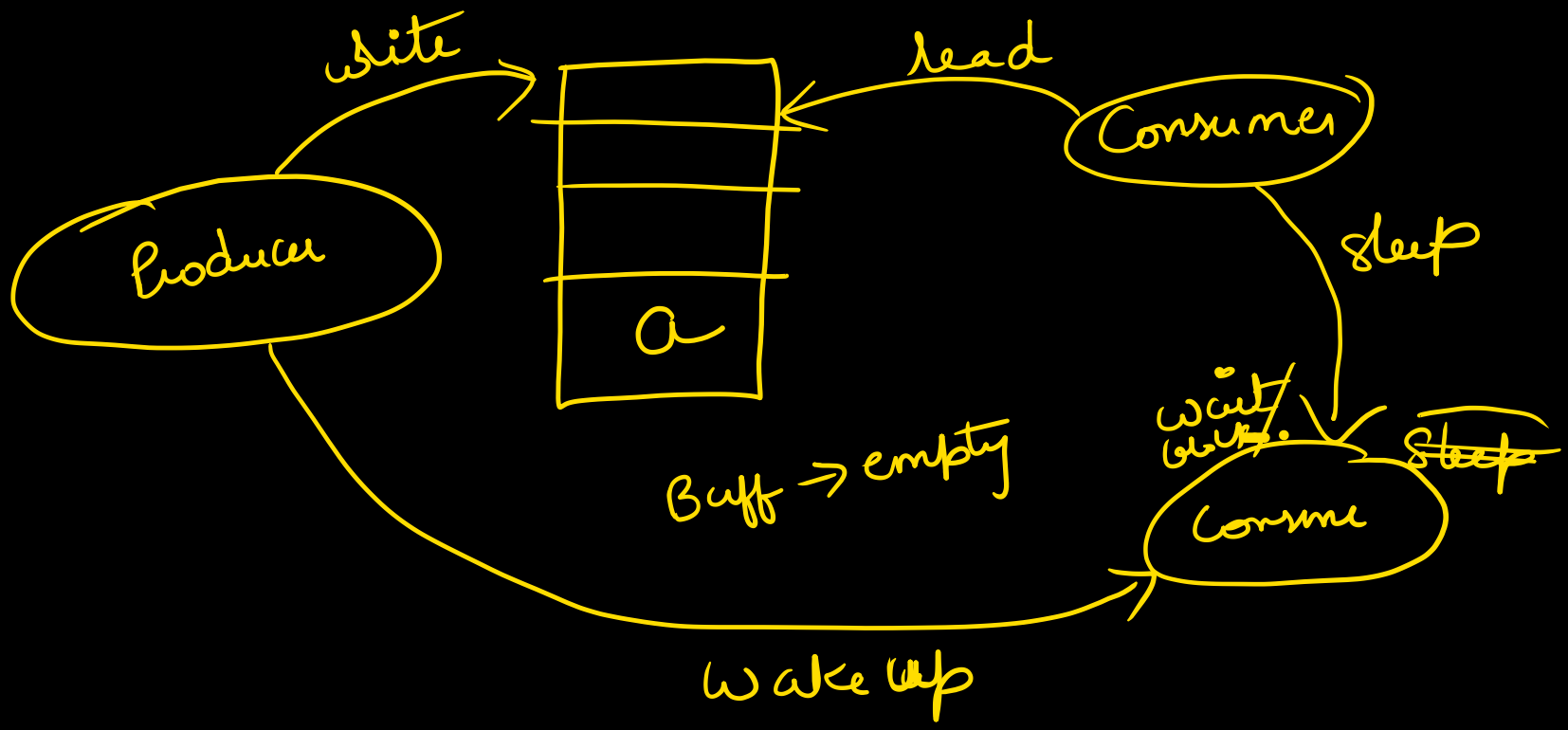
(No Busy waiting) > Busy waiting

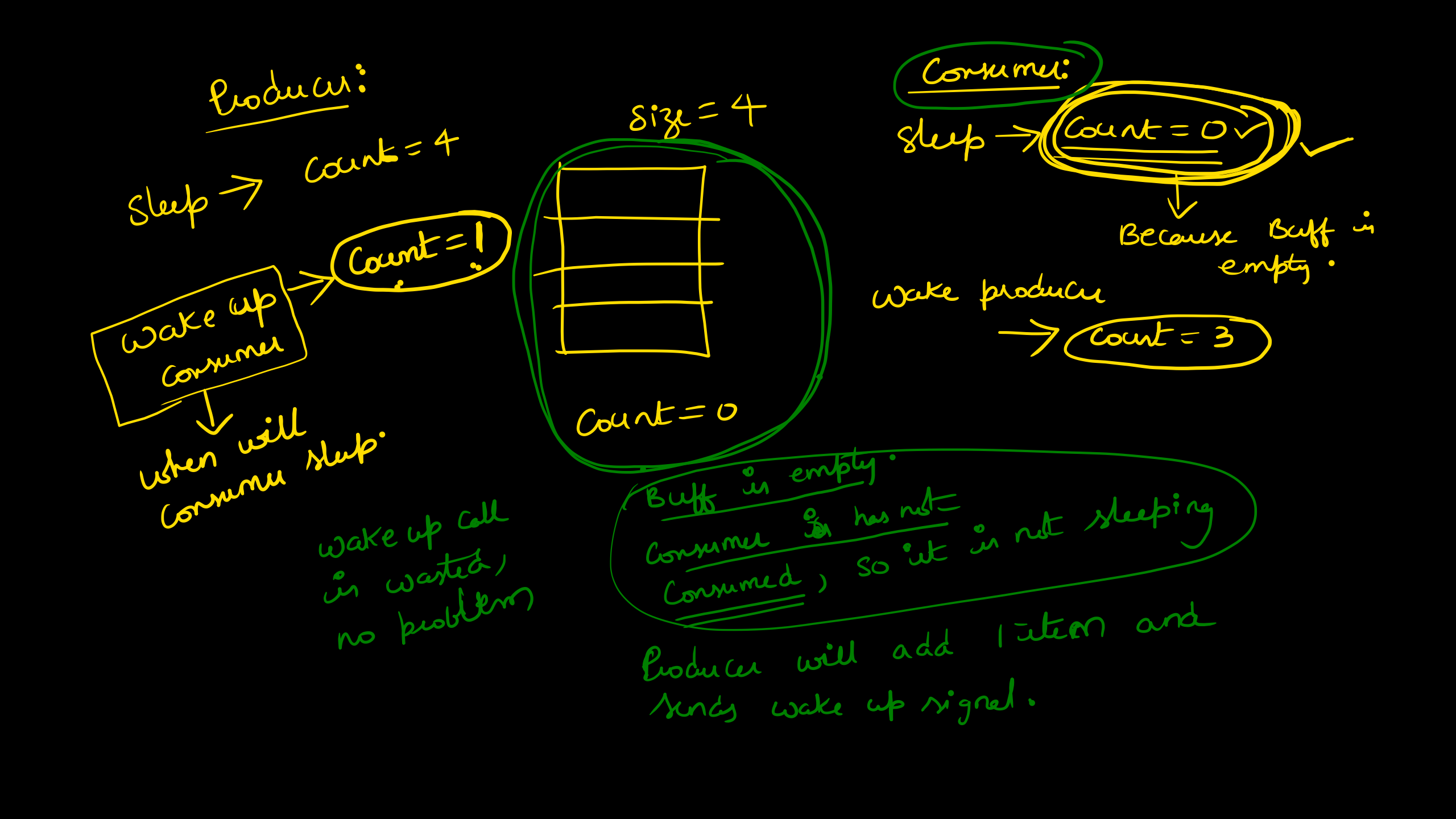


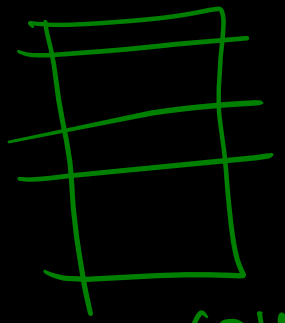
their turn will come.
they dont wait forever.

Sleep and wake up:




write
read
Producer
Consumer
sleep
a
Buff → empty
wait
Consume
wake up

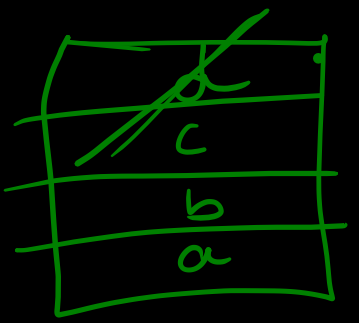

Producer:
Sleep →
count = 4
Wake up consumer
Count = 1
When will consumer sleep.
size = 4
Count = 0
Consumer:
Sleep →
Count = 0
Because Buff is empty.
Wake producer
→ Count = 3
Wake up call is wasted, no problem
Buff is empty.
Consumer is has not Consumed, so it is not sleeping
Producer will add 1 item and sends wake up signal.



count = 0.

Consumer may or may not be sleeping.

But producer will send wake up at count = 1 with a safe assumption that consumer is sleeping..



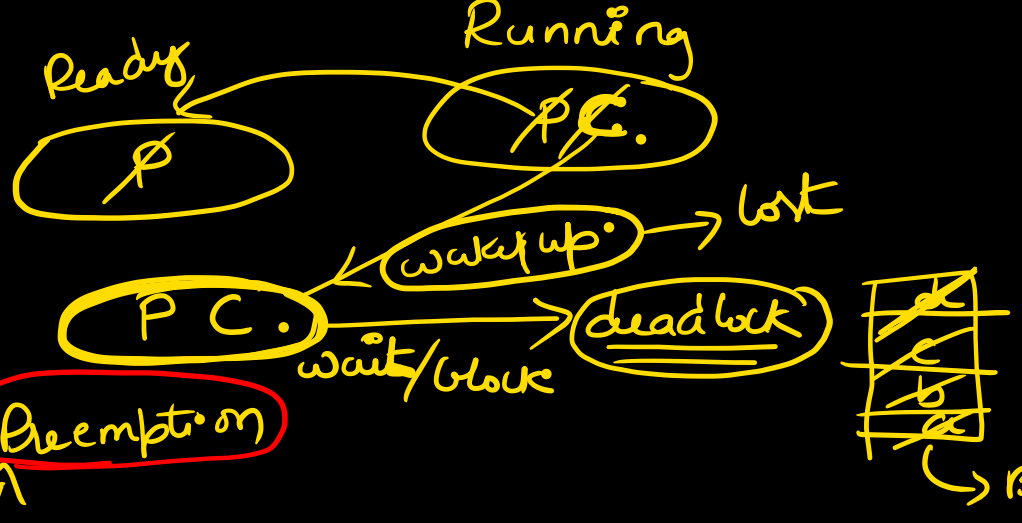
Buff full.

If (producer) attempts to write then it will sleep but it did'nt attempt (lets say) then it is not sleeping.

But Consume will sent wakeup when count = 3.

Safe

```
# define N 100
# define count 0
void producer (void)
{
    int item;
    while (TRUE)
    {
        item = produce_item();
        if (count == N) sleep();
        insert_item (item);
        count ++;
        if (count == 1) wake up (consumer);
    }
}
```



Buff is full.

Buff is full

consumer maybe sleeping
So wakeup

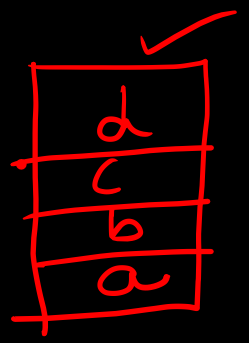
break = 5 min

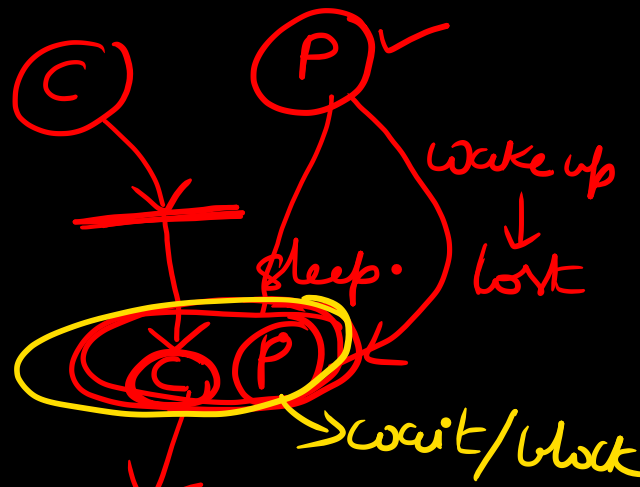
```
void consumer (void)
{
    int item;
    while (TRUE)
    {
        if (count == 0) sleep();
        item = remove_item()
        count --;
        if (count == N-1)
            wake up (producer);
        consume_item(item);
    }
}
```

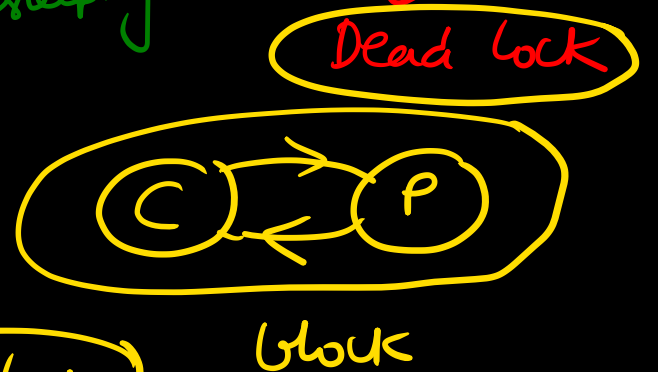
preemption

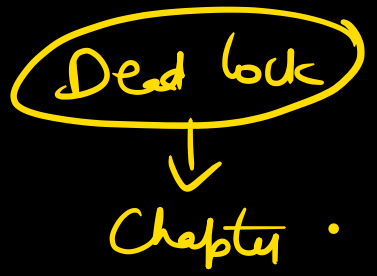
Buff empty.

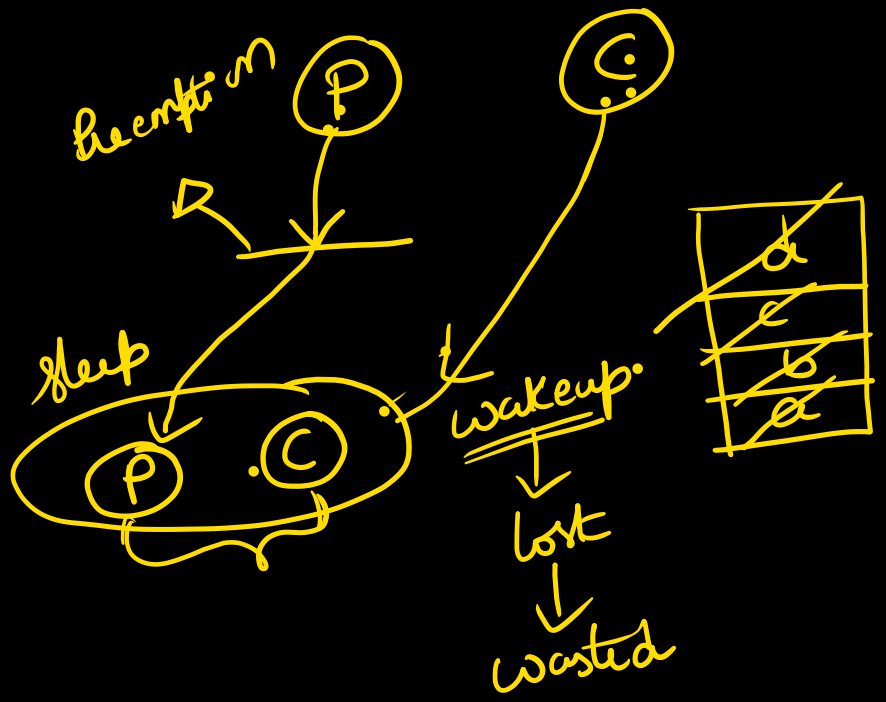
Producer maybe sleeping



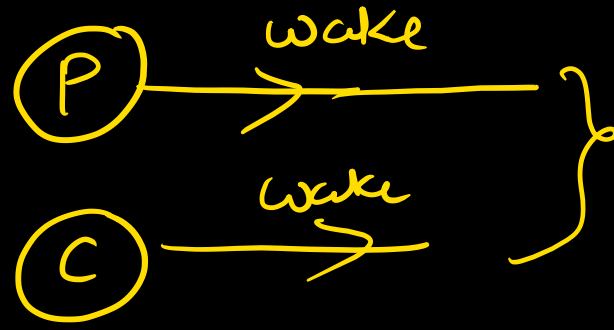






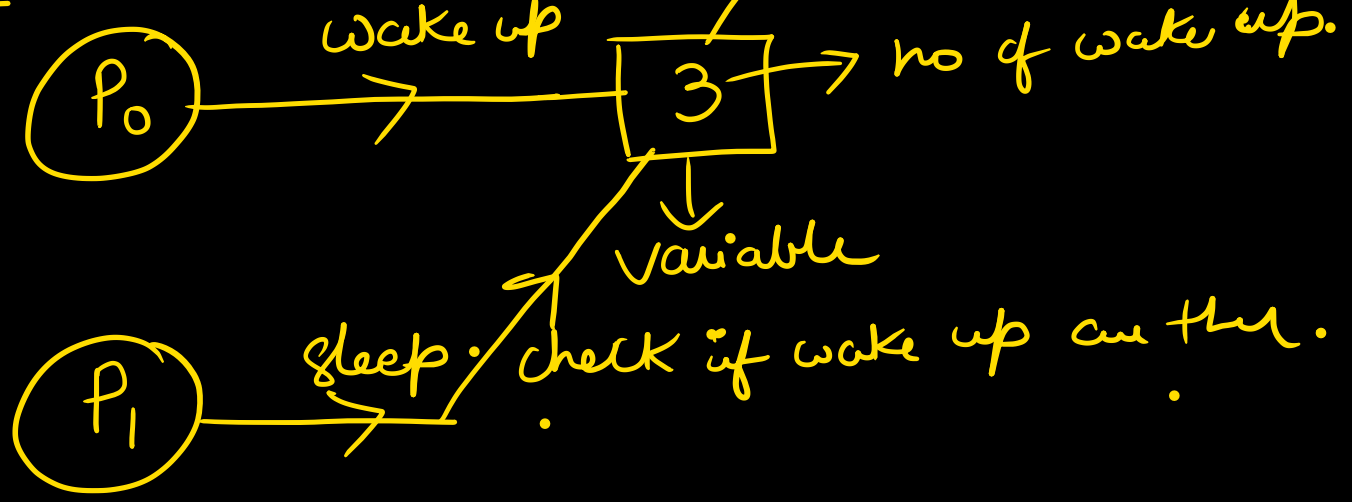

preemption
P
C
sleep
P
C
wakeup
lost
wasted
d
c
b
a

Problem with sleep/wake up.



Concept behind semaphore. ✓

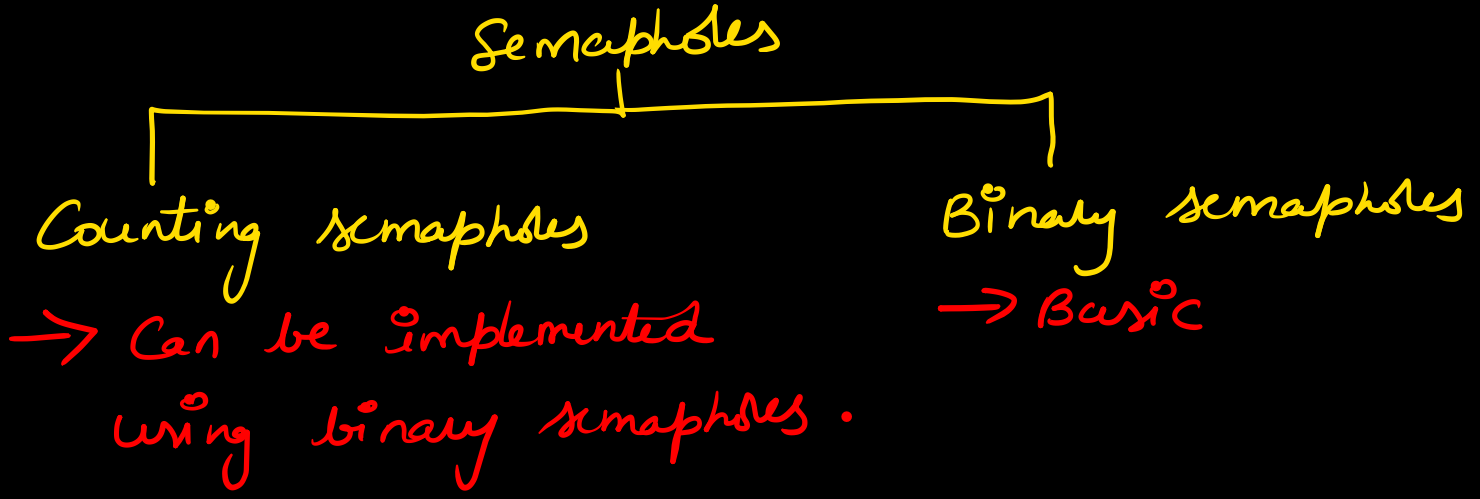
Solution:



Semaphores:

Variable on which Read and update happens atomically in Kernel mode.

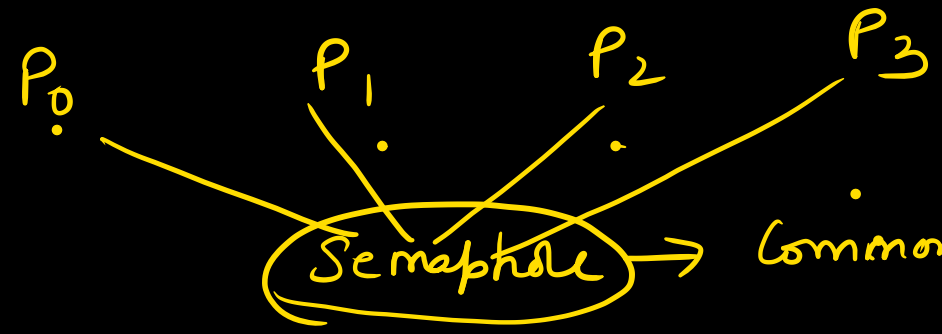
→ Support from OS is required



Counting semaphores:

These are used when the resources (Critical section) can be used by more than one process simultaneously.

Ex: If 4 processes can enter CS simultaneously, then ~~CS~~ Counting semaphore $\boxed{a = 4}$.



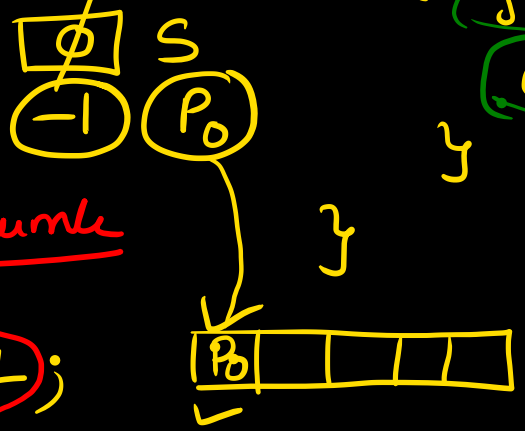
```
Struct Semaphore
{
    int value;
    Queue type L;
}
```

+ve → how many Processes can enter CS

-ve → how many Processes are blocked

```
Down (Semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        Put Process (PCB) in L;
        sleep();
    }
    else
        return;
}
```

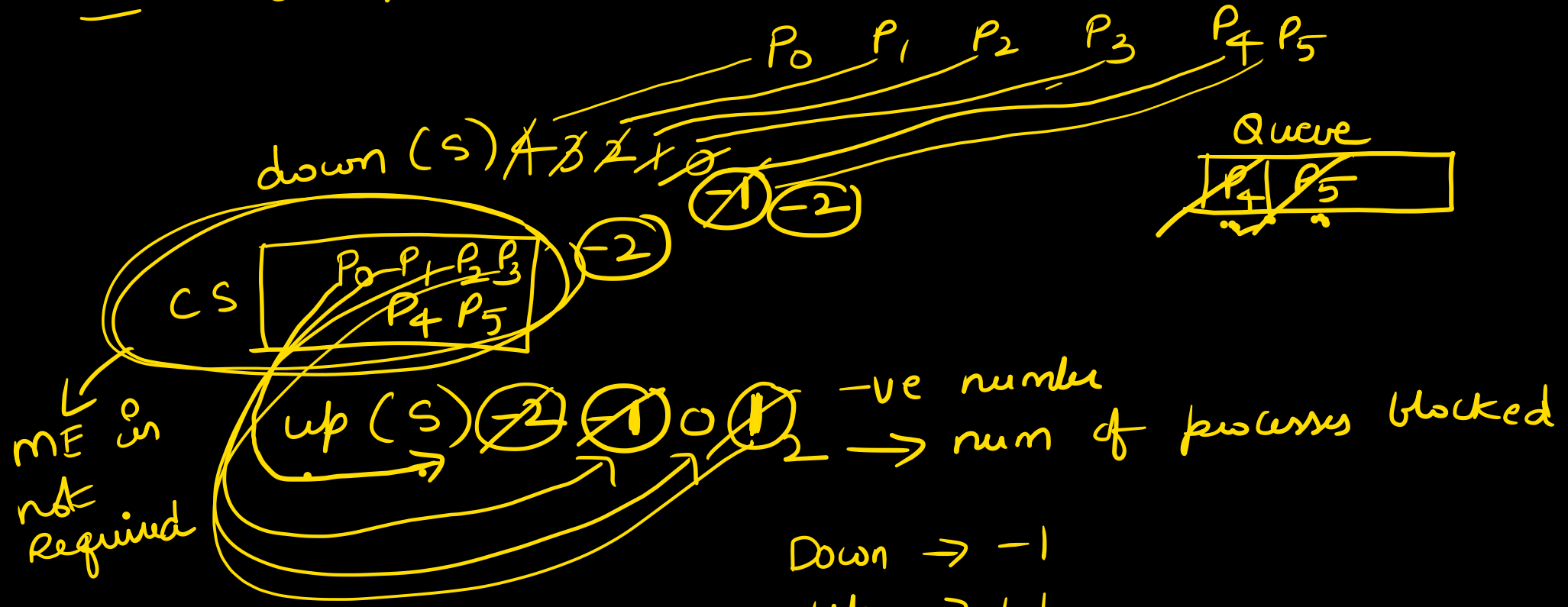
-ve number




```
up (Semaphore S)
{
    S.value = S.value + 1;
    if (S.value ≤ 0)
    {
        select a process from L;
        wakeup();
    }
}
```

Before incrementing -ve, → Some process are sleeping.

Ex: Semaphore S = 4 .



ME → depends on whether we want it.

If S = 4 → 4 process can enter CS

if S = 1 → only one process → ME

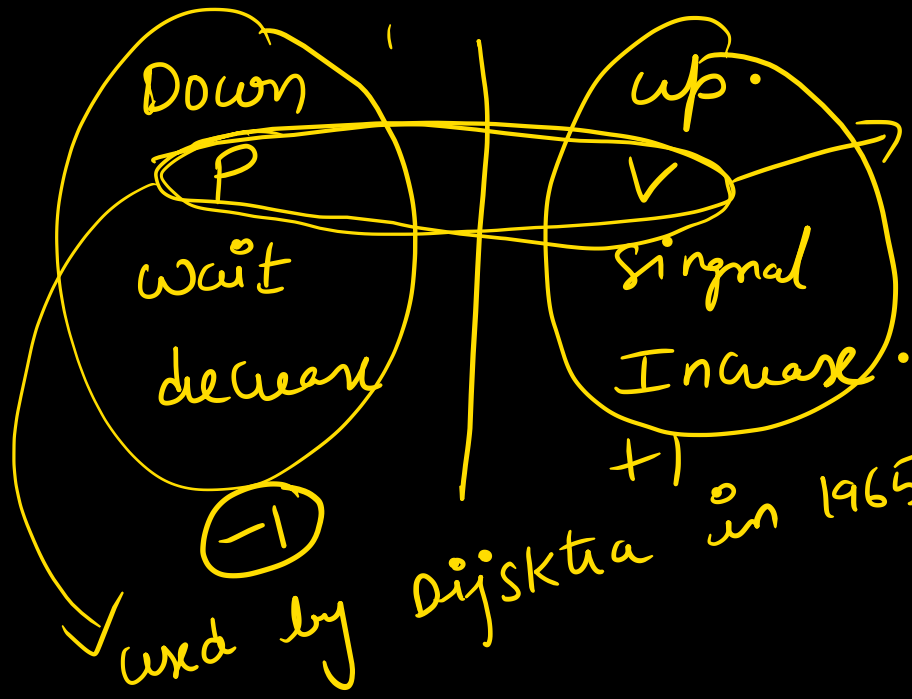
Progress → No process can stop other process.

Bounded wait → we are using a queue (FIFO)

every process will get a chance in FIFO order.

Architectural neutral → ✗

↳ dependent on OS.



very popular.

used by Dijsktra in 1965 paper.

98) A counting semaphore was initialized to 10. Then (6P) and (4V) operations were performed. What is result?

$$10 - 6 + 4 = 8$$

92) $S = 7$, 20 P, 15 V $S = ?$

$$7 - 20 + 15 = 2.$$