CS & IT ENGINEERING

Operating System

Process Synchronization /Coordination

Lecture No. 4

By- Dr. Khaleel Khan sir

- Synchronization Hardware
- TSL & SWAP Instructions
- Sleep-Waakeup

Q.

Two processes, P_1 and P_2, need to access a critical section of code. Consider the following synchronization construct used by the processes:

```
/*P1*/
    while (true)
    {
            wants1 = true;
            while (wants2 == true);
            / * Critical Section * /
            wants 1 = false;
    }
    / * Remainder section * /
```

```
/*P2*/
    while (true)
    {
      wants2 = true;
      while (wants1 == true);
      / * Critical Section * /
            wants 2 = false;
    }
    / * Remainder section * /
```

Here, wants1 and wants2 are shared variables/ which are initialized to false. Which one of the following statements is TRUE about the above construct?

A. It does not ensure mutual exclusion.

B. It does not ensure bounded waiting.

C. It requires that processes enter the critical section in strict alternation.

D. It does not prevent deadlocks but ensures mutual exclusion.

Q. Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes

Process X

```
/* other code for process X */
while (true)
{
    varP = true;
    while (varQ == true)
    {
            /* critical section */
            varP = false;
    }
}
/* other code for process X */
```

Process Y

```
/* other code for process Y */
while (true)
{
    varQ = true;
    while (varP == true)
    {
        /* critical section */
        varQ = false;
    }
}
/* other code for process Y */
```

(Cont.....)

Q. Here, varP and varQ are shared variables and both are initialized to false. Which one of the following statements is true?

A. The proposed solution prevents deadlock but fails to guarantee mutual exclusion

B. The proposed solution guarantees mutual exclusion but fails to prevent deadlock

C. The proposed solution guarantees mutual exclusion and prevents deadlock

D. The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion

II. Synchronization Hardware

(i) TSL (Test & Set lock) (ii) SWAP } Lock-based Mechanism

→ Busy waiting
→ H/w Soln
→ Multi-Process Soln

---

① T.S.L (Test & Set lock)

TSL(&lock)

⇒ Any Process executing TSL instruction, returns the Current value of lock & Sets the value of lock to "TRUE" (Atomic)

RQ

P1	P2	P3	. . .

FALSE lock

CPU

<CS>

```
Bool TSL(Bool *flag)
{
    Bool rv;
1.  rv = *flag;
2.  *flag = TRUE;
3.  return(rv);
}
```

Atomic Execution

```
Bool Lock = FALSE;

Void Process(i)
{
  while(1)
  {
    a) Non_CS();
Entry  b) while(TSL(&lock)==TRUE);
    c) <CS>
Exit   d) Lock = FALSE;
  }
}
```

(i) Mut. Exclusion: ✓

(ii) Progress: ✓

(iii) Bounded wait:

"This given implementation of TSL, does not guarantee Bounded wait"

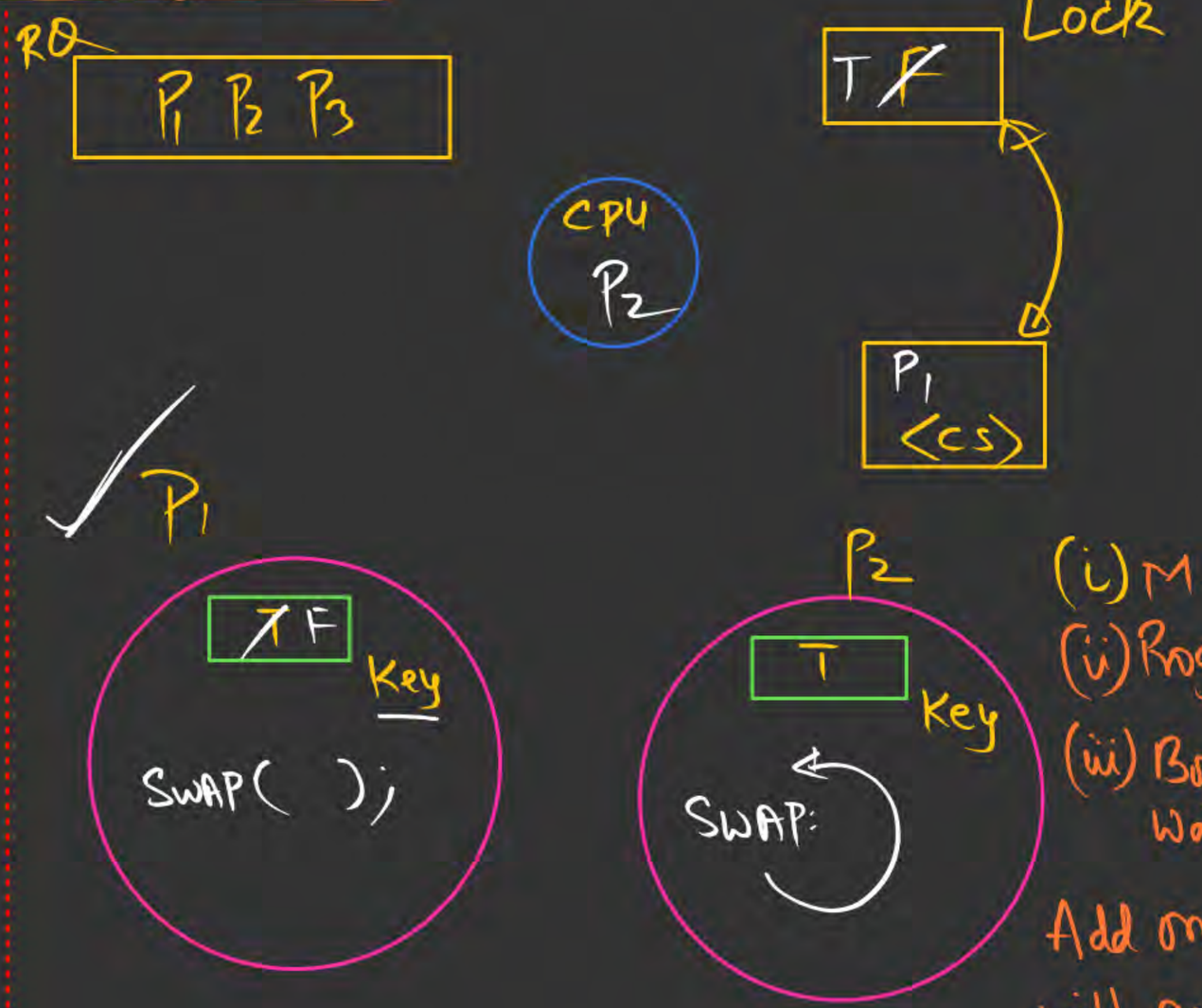
Text Book of Galvin,

"Additional logic is added in entry section to guarantee Bound. Wait"

(iii) SWAP - Instruction: Lock - key based

```
SWAP(Bool *a, Bool *b)
{
    Bool t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Atomic Execution



(i) M.E: ✓
(ii) Progress: ✓
(iii) Bound: ✗ Wait

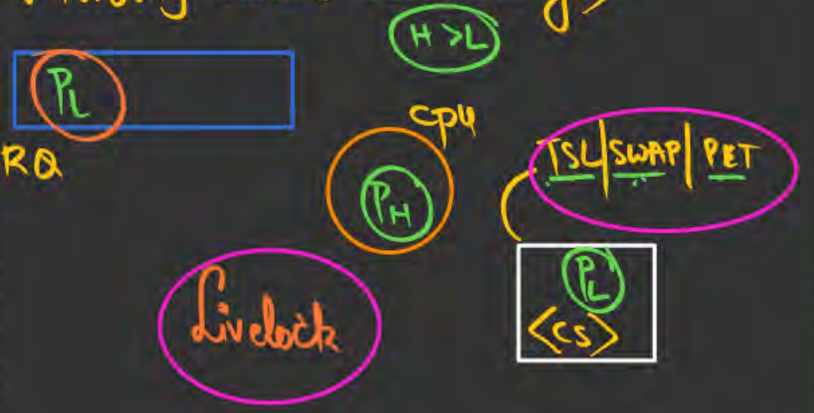
Add on logic will guarantee

```
Bool Lock = FALSE;
void Process(int i)  i=1, n
{
    Bool Key;
    while(1)
    {
        a) Non-CS();
        b) Key = TRUE;
        c) do
           {
               SWAP(&Lock, &Key);
           } while (Key == TRUE);
        d) <CS>
        e) Lock = FALSE;
    }
}
```

Limitations of TSL & SWAP Instns

1) Busy-waiting:
   ↳ wastage of cpu time

2) Priority Inversion Problem:
   : Priority based Scheduling;



Priority Inheritance Protocol

Q.

The enter_CS ( ) and leave_CS () functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
while (test-and-set(X));
}
void  leave_CS(X)
{
X=0;
}
```

does not guarantee Bounded wait

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?

A. I only

B. I and II

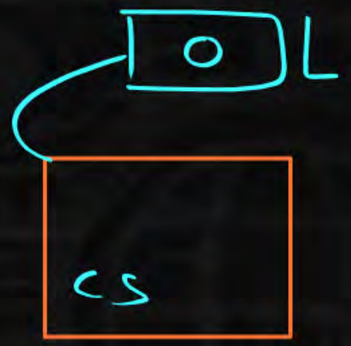
C. II and III

D. IV only

Q.

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.

```
AcquireLock(L)
{
  While (Fetch_And_Add (L,1))
  L=1;
}
ReleaseLock(L)
{
  L=0;
}
```




```
Fetch_Add(x,i)
{
    int ret;
Atomic{ ret = x;
        x = x+i;
        return(ret);
      }
```

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

III. OS-based: (if-then-else) < Blocking / without Busy-waiting > (to avoid wastage of cpu-time, in the loop)

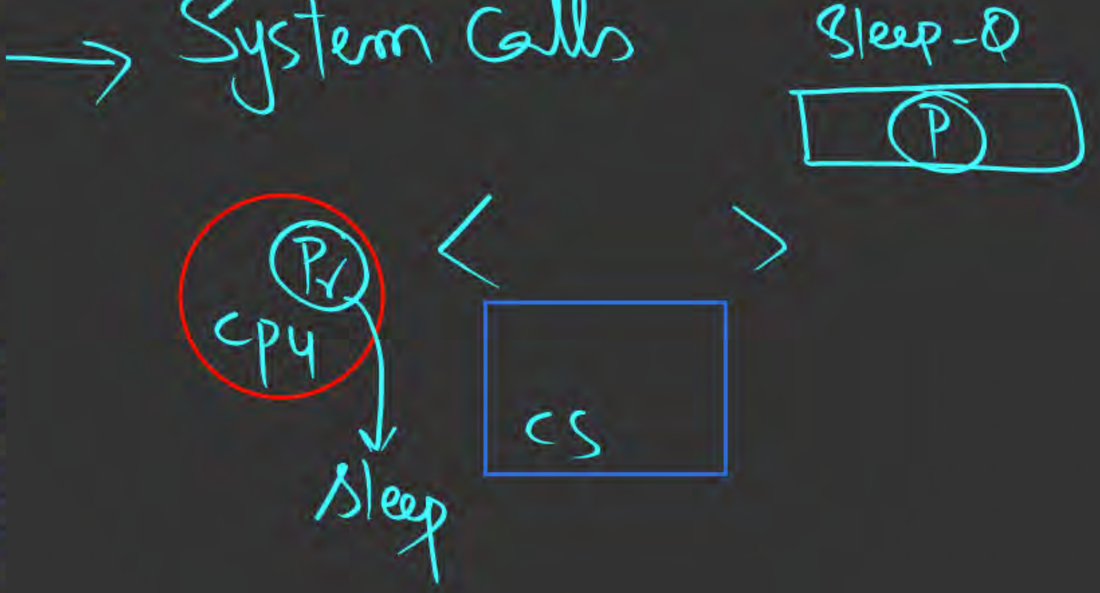
(i) Sleep() - wakeup()

*(ii) SEMAPHORES

(iii) MONITORS

Sleep-wakeup

→ System Calls



II. Producer-Consumer using Sleep & Wakeup :

```
#define N 100
int Buffer[N];
int Count = 0;
```

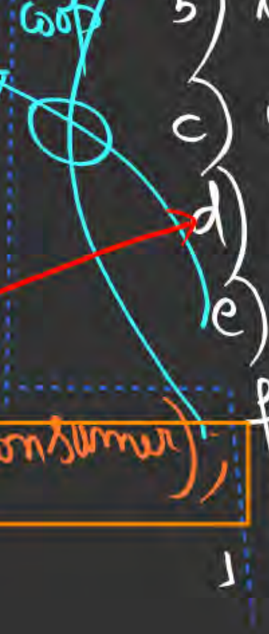
```
void Producer(void)
{
    int itemp, in = 0;
    while (1)
    {
a)      itemp = Produce_item();
b)      if (Count == N) Sleep();
c)      Buffer[in] = itemp;
d)      in = (in+1) % N;
e)      Count = Count + 1;
f)      if (count == 1) wakeup(Consumer);
    }
}
```

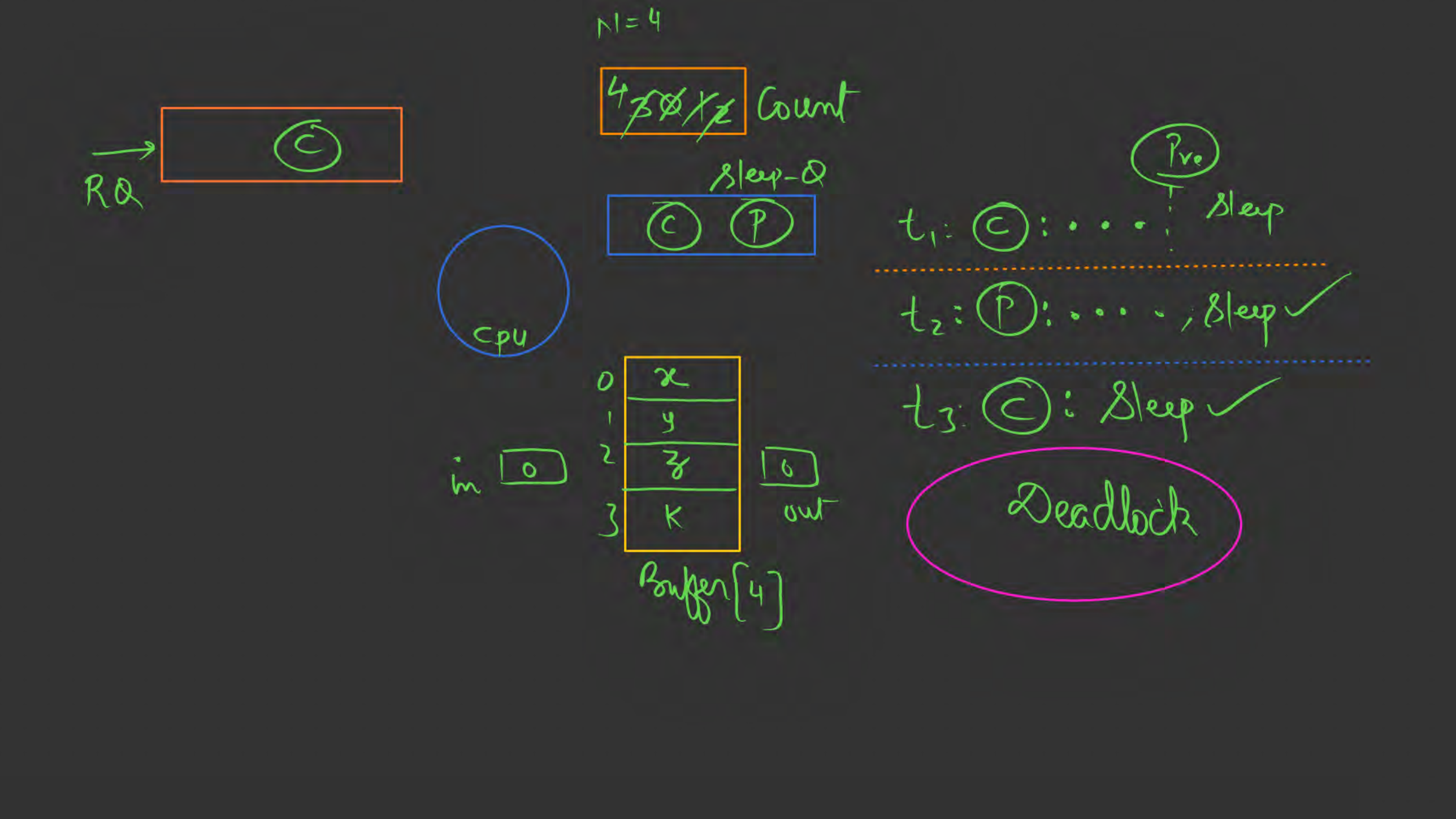
```
void Consumer(void)
{
    int itemc, out = 0;
    while (1)
    {
a)      if (Count == 0) Sleep();
b)      itemc = Buffer[out];
c)      out = (out + 1) % N;
d)      Count = Count - 1;
e)      if (Count == N-1) wakeup(Producer);
f)      Process_item(itemc);
    }
}
```

Pre

(i) Inconsistency

(ii) Deadlock ?




N1=4
4 3 2 1 0 Count
RQ
C
Sleep-Q
C
P
cpu
Pre
t1: C : . . . . Sleep
t2: P : . . . . , Sleep ✓
t3: C : Sleep ✓
in
0
0
x
y
z
k
out
Buffer[4]
Deadlock


THANK YOU
GW
SOLDIERS !