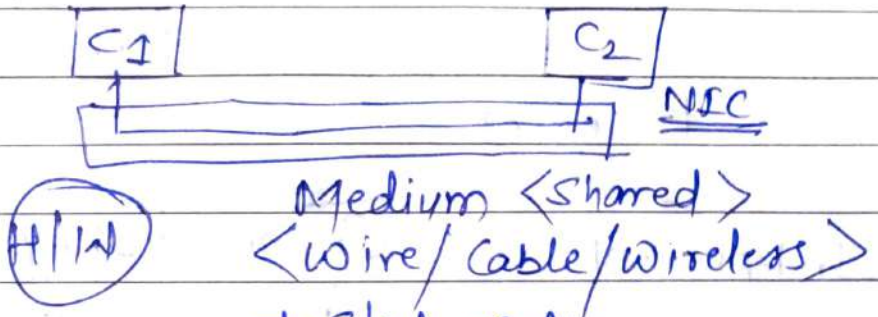
UNIT-III

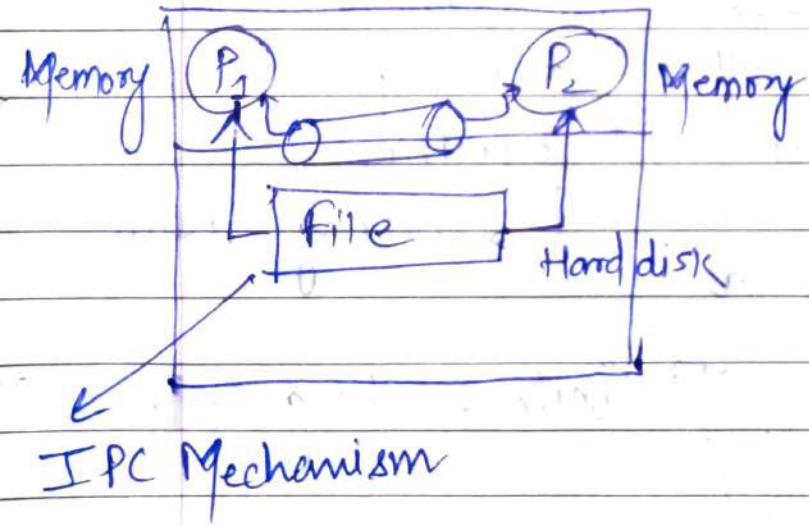
** IPC & Process Synchronization

↓

<Inter Process Communication>

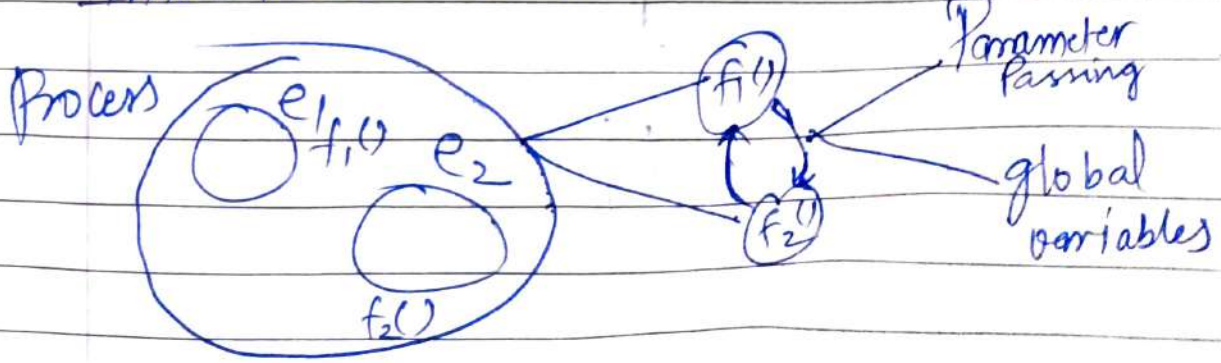


H/W + S/W → Media



IPC Mechanism

Intra Process Communication



Processes

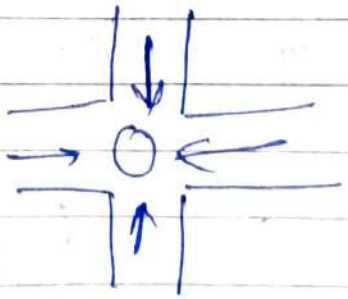
Independent

Communicating
Co-ordinating

Synchronization :

Lack of Synch in IPC Environment leads to the following :

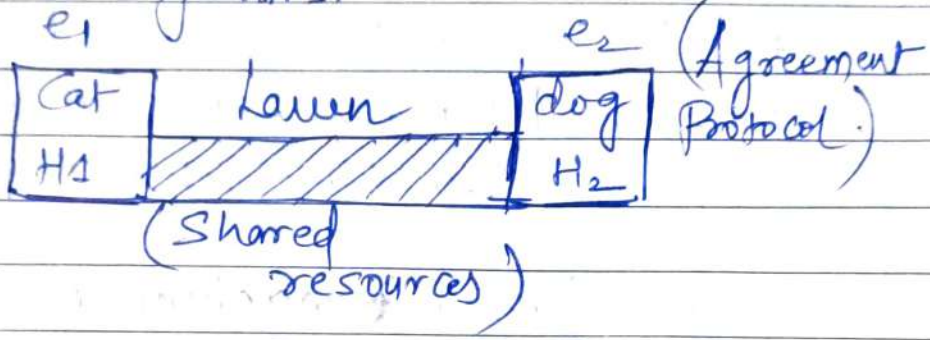
1. Inconsistency [Incorrectness]
   ↳ Wrong result
2. Loss of data
3. Deadlock [infinite blocking of Process]
   [Lockup]



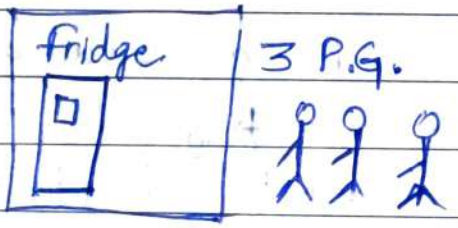
Need for Synchronization

1. Never done in Inconsistency
2) Never loss of data
3) Process Never be in Deadlock.

Ex: 1) Sharing a lawn among Neighbours.



2) More Milk Problem: Paying Guest



Synchronization:- It is agreed upon Coordination Protocol in a IPC environment, to make sure that there is no Inconsistency, data loss or deadlock.

Process Synchronization

Synchronization involves the orderly sharing of system resources by processes

We can think of this intersection as a system resource that is shared by two processes

* The Car processes and the train process
* One process is active at a time – No
* Both process are active – Conflict. Conflict

Consider a machine with a single printer:

Depending upon whether the printer is already being used by another process or not, the operating system must decide whether.

* To grant the printing request (if the printer is free),
* or to deny the request and classify a process as a waiting process until the printer available

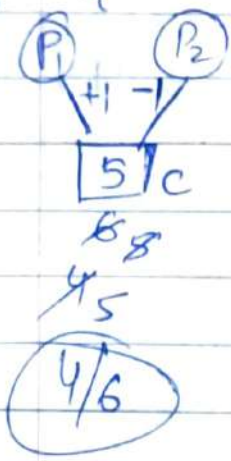
⇒ Types of Synchronization :-

1) Competitive

2) Co-operative

Two/more Processes are said to be in Competitive Synchronization iff they Compete/Contend for the accessibility of a shared resources

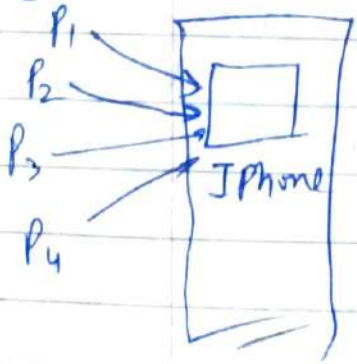
Ex:



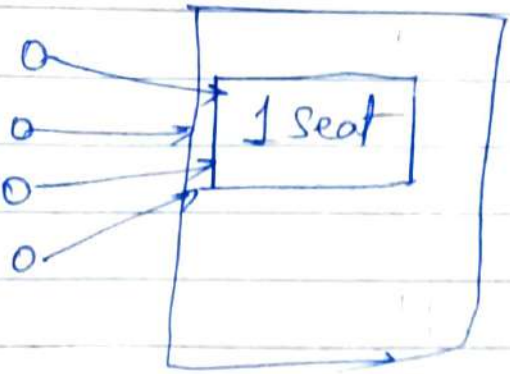
Inconsistency

Example of Competitive Synch.

1) Amazon Purchase

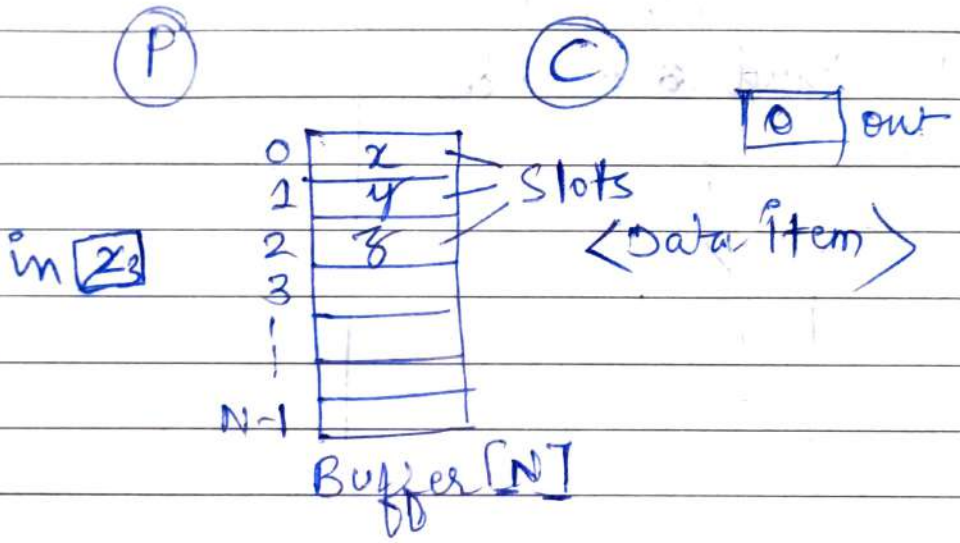


2) Reservation System
Airline/Railway



Cooperative :- Two or more Processes are said to be in Cooperative Synch. iff they get affected by each other ; i.e. execution of one process affects the other Process <Dependency>

Ex: Producer Consumer Problem.



Condition :-

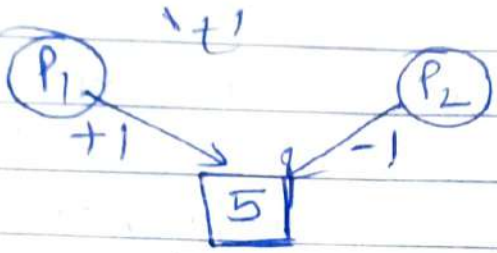
(i) full Buffer
(ii) Empty Buffer

Note1 :- Lack of Synch. among Competing Processes may lead to either inconsistency, Data loss

Note2 :- Lack of Synch among Co-operating Processes may lead to deadlock.

Note 3:- An application of an IPC environment may involve either Competition (or) Cooperation (or) Both types of synch.

Ex: To prove inconsistency



Consistent Final Consistency Value must be 5 4

Sometimes, it is possible to get 5 and other times it is also possible to get 4 or 6

```
{
C = C + 1
}
```

```
I. Load R1, C;
II. Inc R1;
III Store C, R1
```

```
{
C = C - 1
}
```

```
I. Load R2, C
II. Dcr R2
III. Store C, R2
```

Scenario:

RQ | P1 | P2 |

CPU

C: 5 4 6

P1

P1 5 6
P2 5 4

Inconsistency

t: P1 I; II; Pr
t1: P2 I; II; III
t2: P1 III

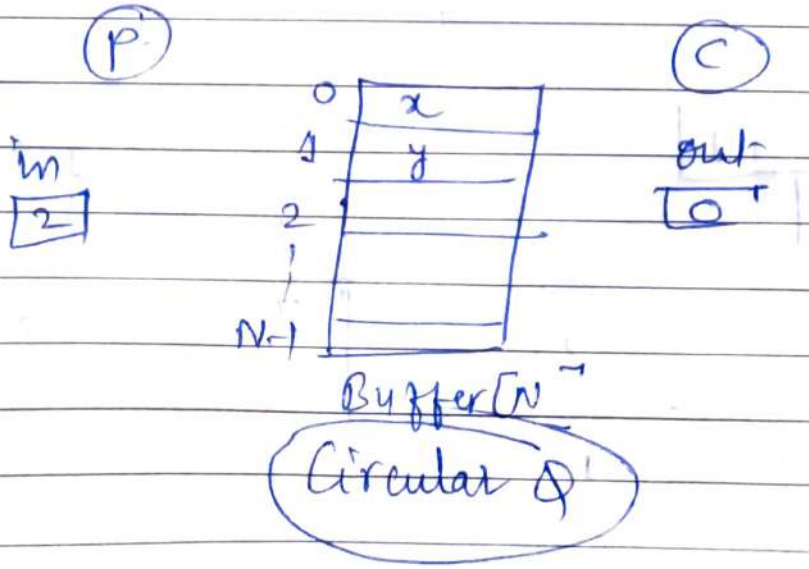
Universal assumption

(Any Process that executes in user mode, can get pre-empted after any instruction)

Some time we get Correct Result & other time, it is possible to get Incorrect results

As a end user we want solution, that gives always Correct result
< whether Preemption takes place or not >

Implementation of Producer Consumer Prob.

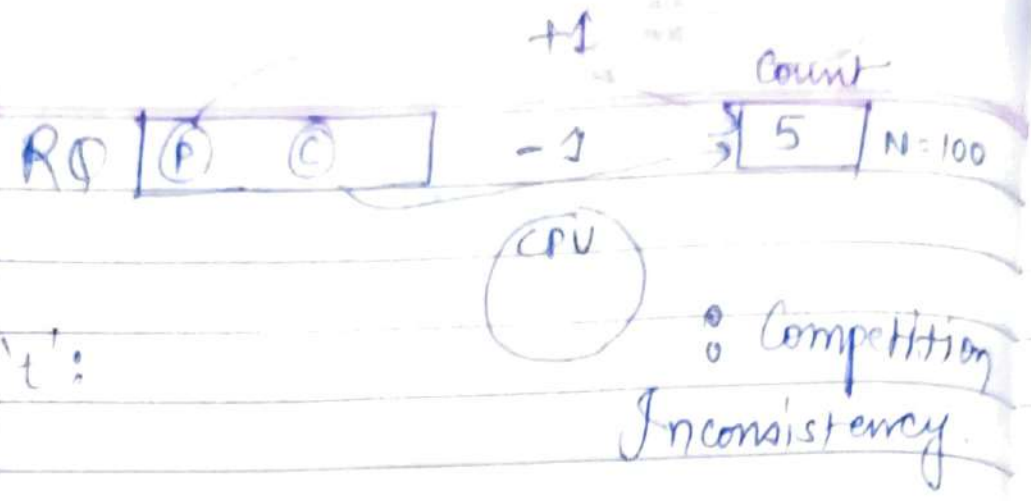


```
#define N 100   int Count=0; (No. of Data item in Buffer)
int Buffer[N];
Void Producer (void)
int itemp, in =0;
while(1)
{
  a) itemp = Produce_item();
  b) while (count == N);
  c) Buffer[in] = itemp
  d) in = (in+1) % N;
  e) Count = Count+1;
}
}
```

Busy waiting

```
void Consumer(void)
{
  int itemC, out=0
  while (1)
  {
    a) while (count == 0);
    b) itemC = Buffer[out];
    c) out = (out+1) % N;
    d) Count = count-1;
    e) Process_item(itemC);
  }
}
```

Busy waiting



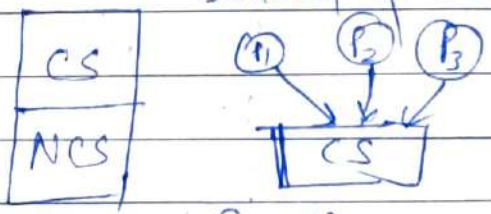
Necessary condition for Synch Problems:

1. Critical section (CS):- is that part of the Program where shared resources are accessed,

Non critical Section (NCS) :- Part of the Program that does not accessed Shared Resources

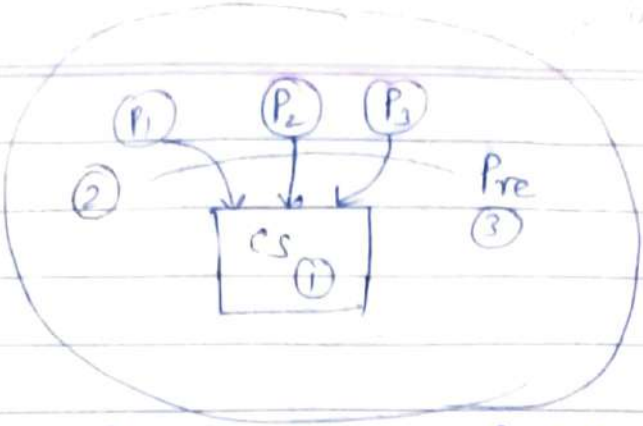
<Other section>

<Remainder section>



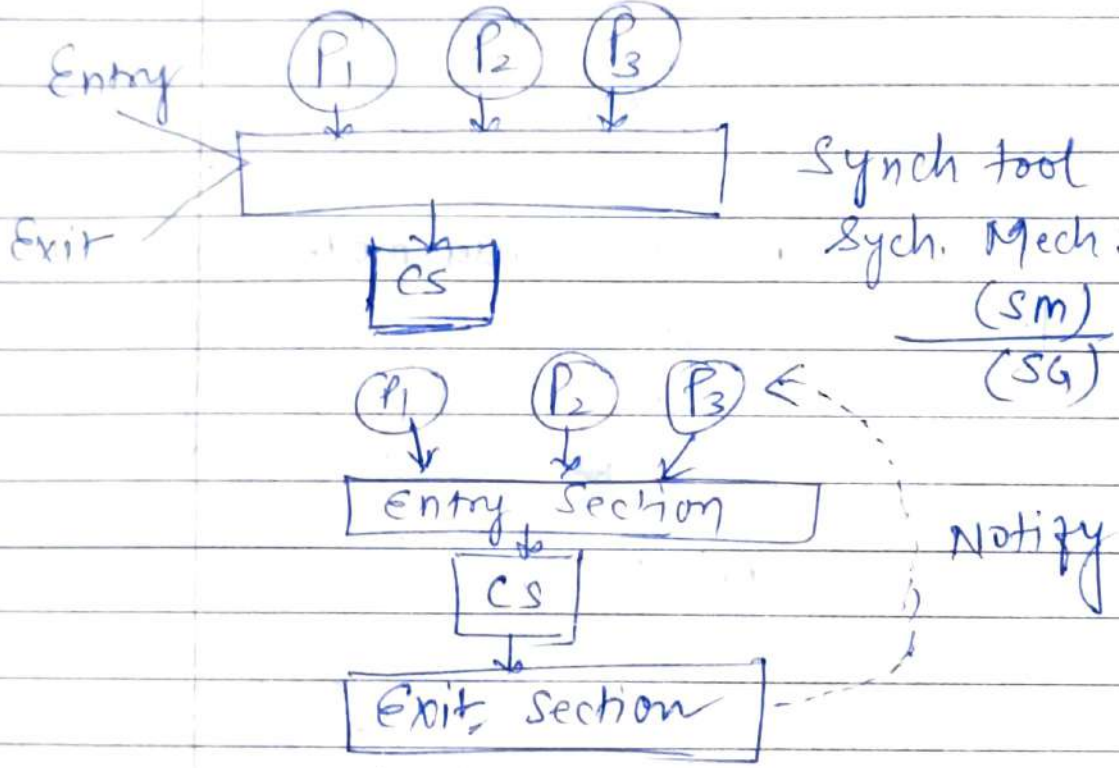
2) Race Condition :- It is situation wherein Processes are trying to access <CS> and final result depend on the order in which they finish their update.

3) Preemption :-



Critical Section Problem

// Solution



<General Architecture / Model of S.M>

Do {

[Entry section]

<Critical section>

[Exit Section]

<Remainder section>

} while (true);

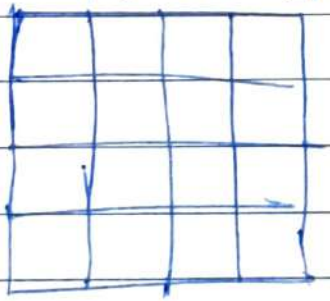
General Structure of a typical Process with Synch. Mechanism

* Process running in user mode can get Preempted any time & any Number of times (After completing any instruction)

Requirements of C.S. Problem

<to be satisfied Synch. Mechanism>

4 queens (q_1 ... q_4)



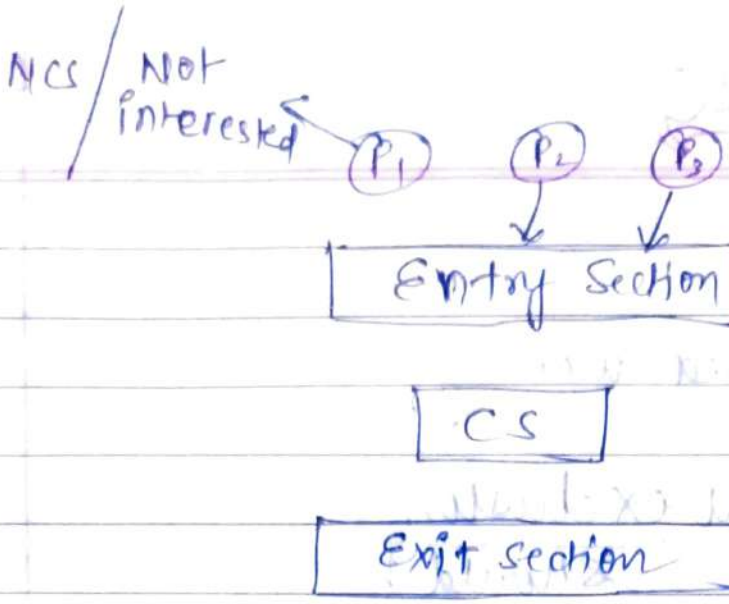
<4-queens>

(i) Mutual exclusion
(ii) Progress
(iii) Bounded waiting

Mutual exclusion :- NO two process should be present in <CS> together at same time.

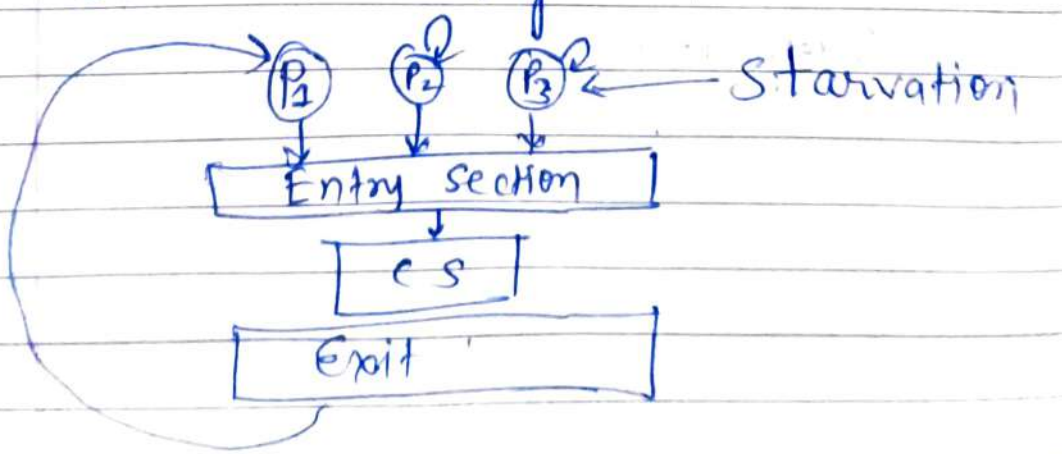
Mutual exclusion :- If Process P_i is executing in its critical section, then NO other processes can be executing in their critical section.

Progress :- If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection can not be postponed indefinitely.



No process (Not interested) has got right to block other processes (Interested) from entering Critical section.

Bounded waiting :- There exists a bound, or limit, on the number of times that other processes are allowed to enter their Critical section after a process has made a request to enter its Critical Section and before that request is granted.



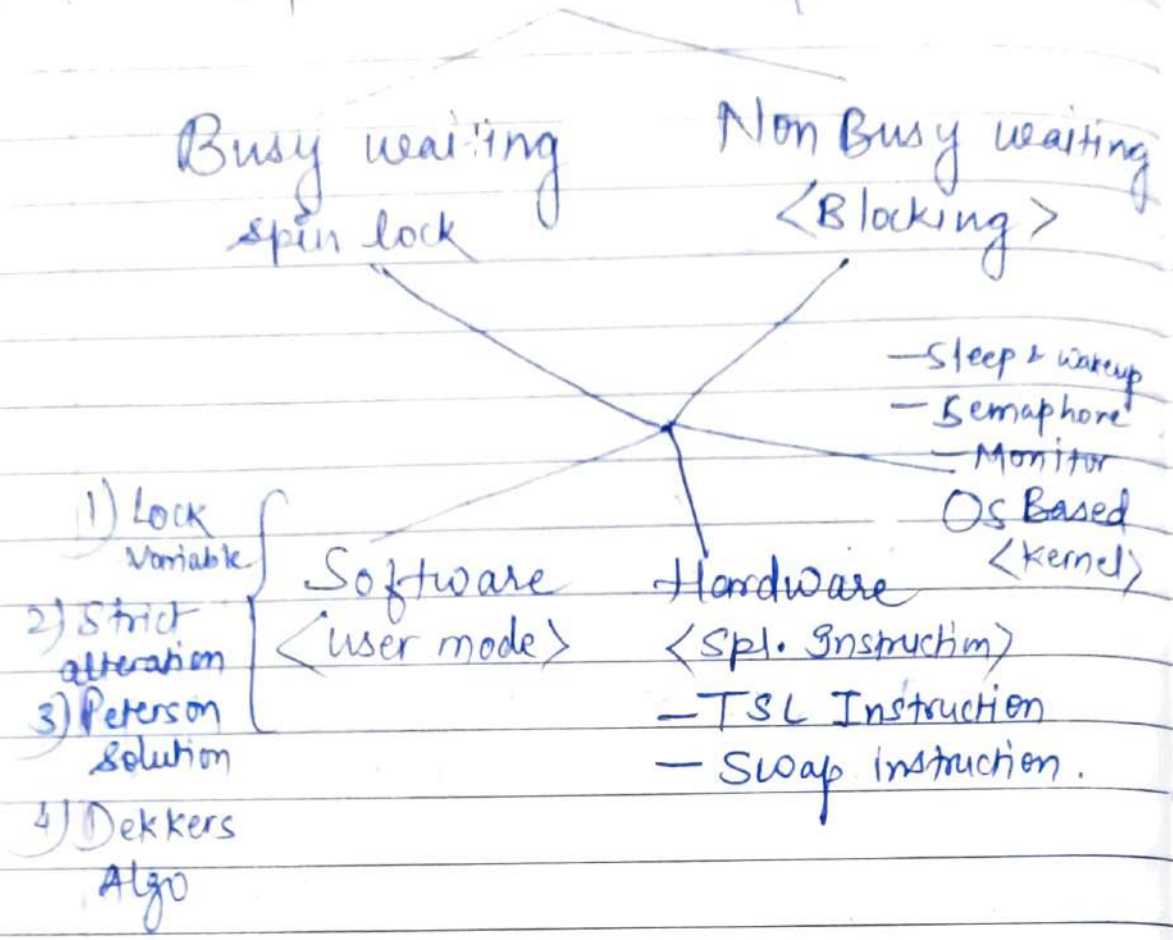
No process has to wait for ever to access Critical Section; their should be a bound on the no. of times a process is allowed to enter Critical Section, before other process ~~required~~ request is granted.

1. If mutual exclusion is not guaranteed then (inconsistency + Lost of Data) may happen.

2. If bounded waiting is not guaranteed then Starvation.

3. If process is violated then it is unfair solution (Indefinite postponement)

Synchronization Mechanism

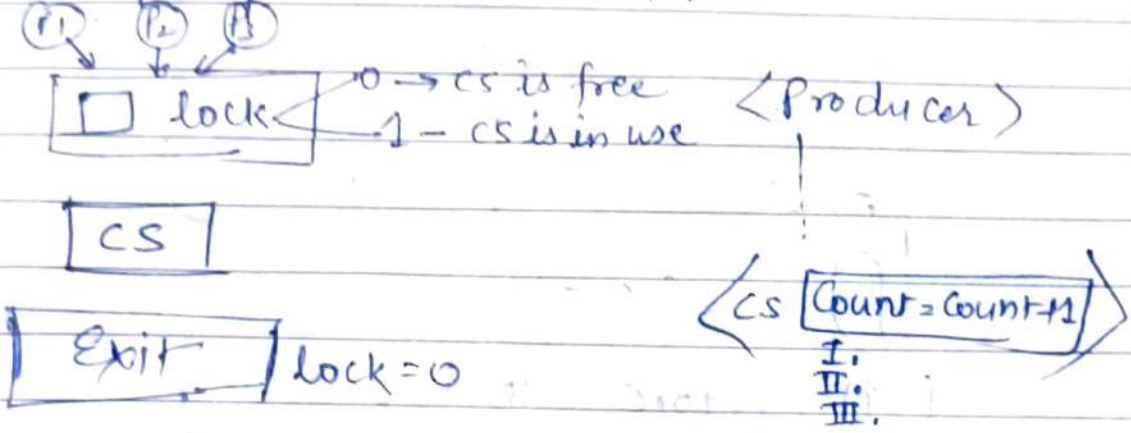


Assumption :-

1) Process enters <cs> and come out of it is in finite amount of time.
2) When a process is in entry section then it means, it is in interested in <cs>
* 3) A Process is said to have left <cs> only when it has executed its exit section.
4) A Process can get preempted from CPU, while executing Entry + <cs> + Exit section;

1) Lock variable :-

→ Busy - waiting solution
→ Software Solution Implementation at user mode
→ Multiprocess solution.



Implementation

```
int lock = 0
Void Process (int i)
{
  while (1)
    a) <Non_cs>
    b) while (lock == 1);      while (lock != 0);
       lock = 1;               Entry section
    c) <cs>
    d) Lock = 0;               Exit section
```

Low Level implementation

integer lock = 0;
Process (int i):

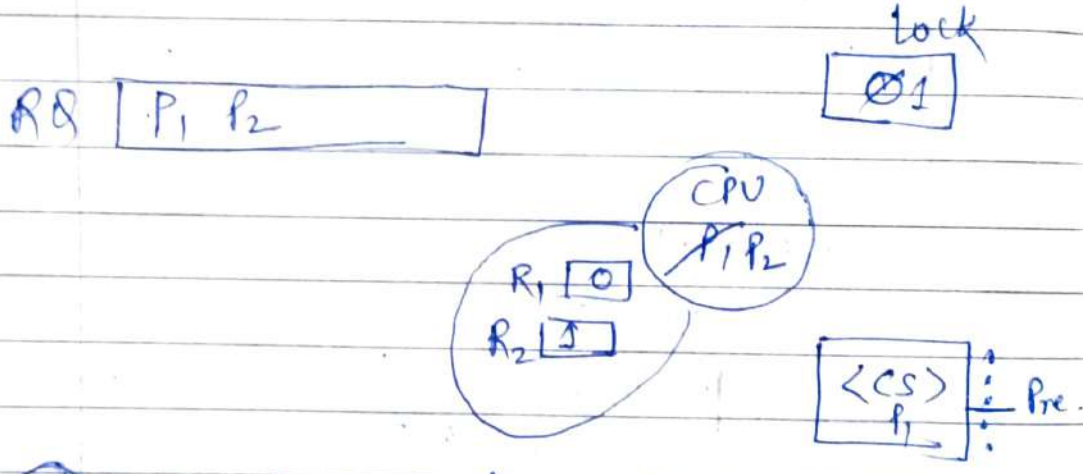
JNZ : Jump if not zero

a) Non_CS
b) Load R_i, lock;
c) Cmp R_i, #0;
d) JNZ step b
e) Store lock, #1;

Entry section

f) <CS>

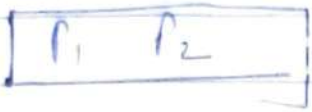
g) Store lock, #0

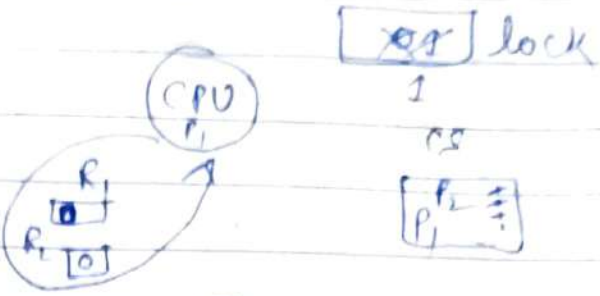


P_1 : <Ncs>: b; c; d; e; f; <cs>; Pre
P_2 : b; c; d; b

→ Does lock variable guarantee Mutual exclusion always ?
→ Progress ?
→ Bounded wait ?

RQ





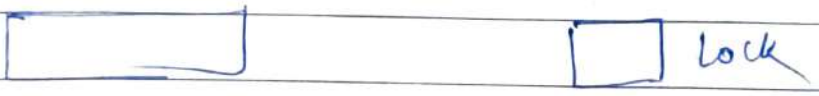
t_1 : (P1) : a; b; c; d; Pre

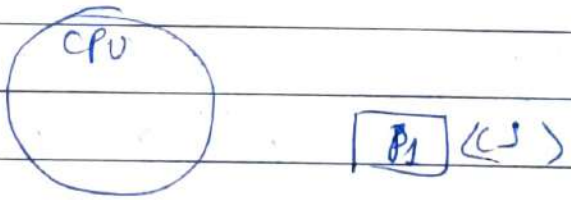
t_2 : (P2) : a; b; c; d; e; f <cs> Pre

t_3 : (P3) : e; f; <cs>

<Mutual exclusion is violated>

<progress is Guaranteed>





t_1 : (P1) : a; b; c; d; e; f; <cs> Pr

t_2 : (P2) : a; b; c; d;) Busy wait; Pre

t_3 : (P1) : <cs>; g; a; b; c; d; e; f;

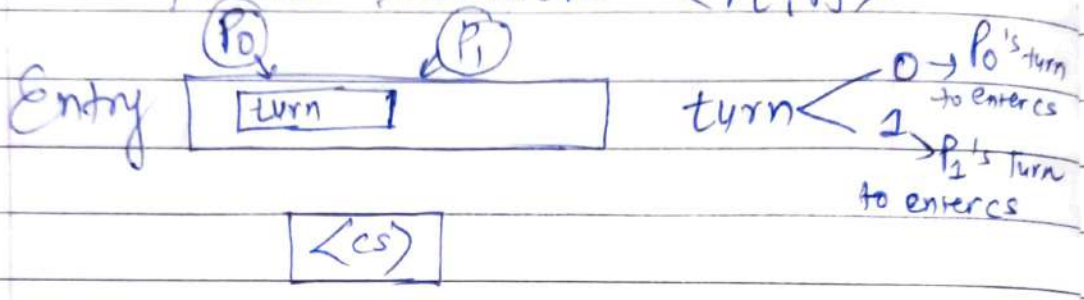
Bounded wait is not guaranteed.

1. Lock variable does not guarantee Mutual exclusion always;
2. Lock variable guarantees progress always.
3. Lock variable fails to Satisfy Bounded waiting (Because a process can enter Critical section multiple times successively while other processes are waiting for their turn to enter '<CS>'.)
4. Lock variable is a busy waiting solution leading to wastage of CPU Time.

Q1 Several Concurrent processes are attempting to share an I/O device. In an attempt to achieve mutual exclusion.

2) Strict Alternation :-

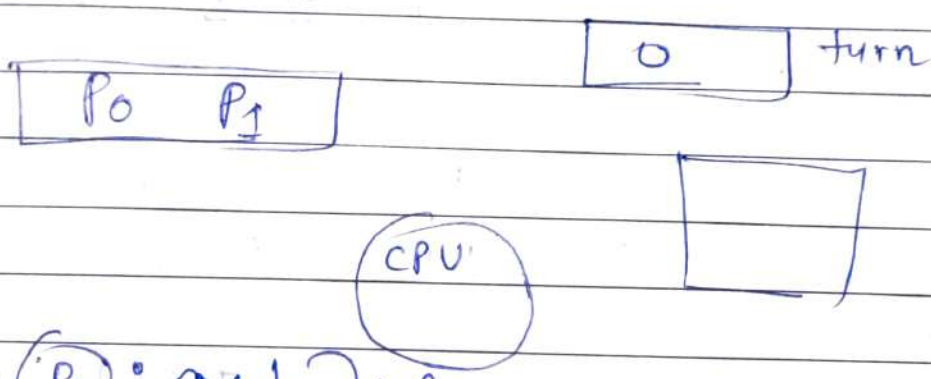
→ Busy - waiting
→ S/w solution implemented @ user mode
→ 2 process solution $\langle P_i | P_j \rangle$



Strictly on alternate basis, processes takes turn to enter critical section

```
int turn = rand(0,1)
void Process(0)
{
    while (1)
    {
        a) Non_cs()
        b) while (turn != 0);
        c) <cs>
        d) turn = 1;
    }
}
```

```
void Process(1)
{
  while(1)
  {
    a) Non_cs();
    b) while(turn != 1);
    c) <cs>
    d) turn=0;
  }
}
```



to (P1) : a; b); Pr
t1 (P0) : a; b;

a) Always guarantee Mutual Exclusion not
b) Progress is guaranteed.

P0: <Non cs>
P1 : a, b,

P0 who is not interested is blocking

(2) Does not Guaranteed Progress.

P_0:

P_1

(3) Always Guarantee Bounded wait.

(4) Busy waiting (wastage of CPU time)

Generalized implementation of strict Alternation :-

```
int turn = rand(i,j);
void Process (int i)
{
    int j = NOT(i)
    while(1)
    {
        a) Non_cs
        b) while (turn != i);
        c) <cs>
        d) turn = j;
    }
}
```

Q Consider the following two process synchronization solution

Process 0

Entry: loop while (turn == 1);
(critical section)
Exit : turn = 1;

Process 1

Entry : loop while (turn == 0);
(critical section)
Exit : turn = 0;

The shared variable turn is initialized to zero. which one of the following is true?

(A) This is a correct two process synchronization solution.

(B) This solution violates mutual exclusion requirement.

(C) This solution violates progress requirements.

(D) This solution violates bounded wait requirement

```
int turn = 0;
void process(0)
{
  while (1)
  {
    a) Non_cs();
    b) while (turn == 1);
    c) <cs>
    d) turn = 1;
  }
}

void process(1)
{
  while (1)
  {
    a) Non_cs();
    b) while (turn == 0);
    c) <cs>
    d) turn = 1;
  }
}
```

Does this guarantee mutual exclusion?

```
int turn = 1;
void process (0)
  {
    while (1)
    {
      a) Non_cs ();
      b) while (turn == 1)
      c) <<s>
      d) turn = 1;
    }
  }
void Process (1)
  {
    while (1)
    {
      a) Non_cs();
      b) while (turn == 1);
      c) <<s>
      d) turn = 1;
    }
}
```

Deadlock

```
int turn = rand(i,j)

void Process (int i)
{
    int j = NOT(i);
    while(1)
    {
        a) Non_CS();
        b) while (turn != i);
           turn = j;
        c) <CS>
        d) turn j;
    }
```

(H/w) (i) Does this guarantee Mutual Exclusion ?

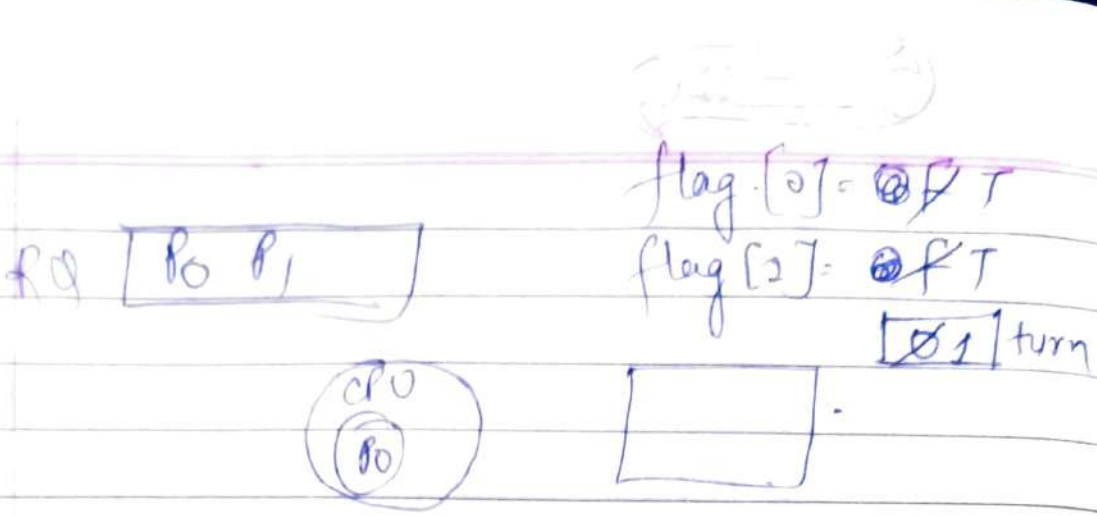
(ii) Does this guarantee Progress ?

(iii) Does this guarantee Bounded wait ?

3) PETERSON'S solution/ALGO

- → Busy waiting
- → Software solution Imple @ u/m
- → 2 process solution
- → Combination of lock variable and Strict alternation.

```
# define N 2
# define TRUE 1
# define FALSE 0

int flag[N] = {FALSE}
int turn;

void process(int i)
{
    int j = NOT[i];
    while(1)
    {
        a) Non_CS();
        b) flag[i] = TRUE;
        c) turn = i;
        d) while(flag[j] == TRUE && turn == i);
        e) <CS>
        f) flag[i] = FALSE;
```



t_1: P_0 $i=0\ j=1$: a; b; Pre

t_2: P_1 $i=1\ j=0$: a, b; c; d : Pre

t_3: P_0 $i=0\ j=1$: d; e

(i) Guarantee Mutual exclusion
(ii) Guarantee Progress.
(iii) Guarantee Bounded wait.

Peterson solution

- 2 Process solution
- Busy waiting (wastage of CPU Time)
- User mode implementation
- Lock variable + Strict Alternation

→ 3 step process

flag[N] = F ;

turn = i/j ;

a) Guarantee Mutual Exclusion
b) Guarantee Progress
c) Guarantee Bounded waiting

Its only disadvantage is that it only works on 2 processes. if more then 2 process then peterson solution failed.

```
void Dekkers_Algorithm(int i)
{
    int j = !(i)
    while(1)
    {
        a) Non_CS();
        b) flag[i] = TRUE;
        c) while(flag[j] == TRUE)
        {
            if(turn == j)
            {
                flag[i] = FALSE;
                while(turn == j);
                flag[i] = TRUE;
            }
        }
        (d) <CS>
        (e) flag[i] = FALSE;
        turn = j;
    }
```

M/E ✓
Progress ✓
B/W ✓

Q Processes P_1 and P_2 use Critical-flag in the following routine to achieve mutual exclusion. Assume that Critical-flag is initialized to false in the main Program.

```
get_exclusive_access()
{
    if (critical_flag == FALSE)
    {
        Critical_flag = TRUE;
        Critical-Region();
        Critical_flag = FALSE;
    }
}
```

Consider the following statements

(i) It is possible for both P_1 and P_2 to access Critical Region Concurrently.

(ii) This may lead to deadlock.

Which of the following holds?

(A) (i) is false and (ii) is true.

(B) Both (i) and (ii) are false.

(C) (i) is true and (ii) is false.

(D) Both (i) and (ii) are true.

Q Two Process 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 */
    wants2 = false;
 }
 /* Remainder Section */

/* P2 */
while (true)
{
   wants2 = true;
   while (wants1 == True);
   /* Critical Section */
        wants2 = 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 process enter the critical section in strict alternation
(D) It does not Prevent deadlocks but ensures Mutual exclusion.

$W_1 = W_2 = F$

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 */
        Variable Q = false;
    }
}
```

Here VarP and varQ are shared variables and both are initialized to false. which one of the following statement 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 prevent deadlock.

(d) The proposed solution does not guarantees mutual exclusion and fails to prevent deadlock.

II SYNCHRONIZATION HARDWARE:

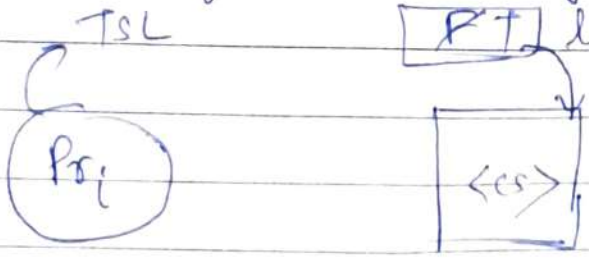
Each processor support some special Instruction (H/W) That are Atomic <Execute Non-Premptive>

a) TSL b) SWAP.

→ Busy waiting
→ ~~Sw solution~~ Can be used at user mode as special instruction
→ Hardware Category.
→ ~~Multi~~ Multiprocess solution
→ lock based solution.

a) TSL: Test & Set lock

TSL(&lock); <Process executing TSL, Returns the current values of lock and sets the value of lock always to true>



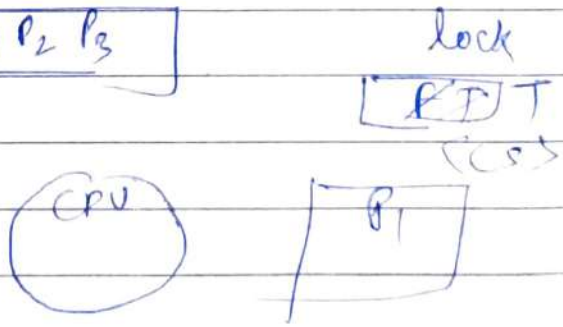
```
Bool TSL(Bool *target)
{
    Bool rv;
    rv = *target;
    *target = TRUE;
    return(rv);
}
```

Atomic execution

Solving CS Problem through TSL.

User prog

```
Bool Lock = F;
Void process(int i)
{
  while(1)
  {
    a) Non_cs();
    b) while(TSL(&Lock)==T);
    c) <CS>
    d) lock = false;
  }
}
```



t_1 : (P1) : a; TSL : F; Pre
t_2 : (P2) : a; TSL : T

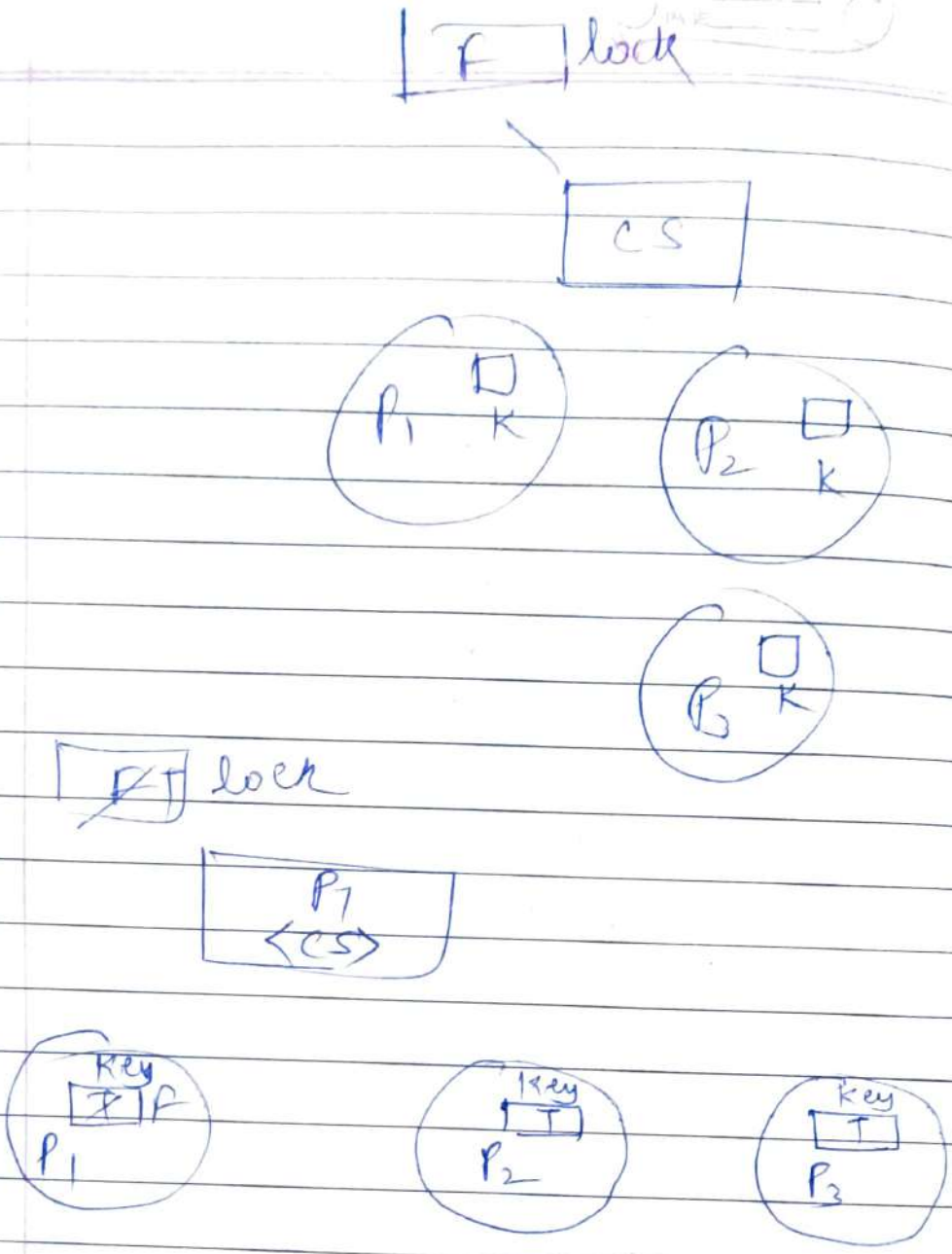
Guarantee Mutual exclusion.
Guarantee Progress.
does not Guarantee Bounded wait.

b) Swap Instruction (ATOMIC)
↳ (lock & key)

```
SWAP(Bool *a, Bool *b)
{
  Bool t;
1. t = *a;
2. *a = *b;
3. *b = t;
}
```

User Process

```
Bool lock = F
void Process(int i)
{
  Bool key = T;
while(1)
  {
    a) Non_cs();
    b) while(key == T)
         SWAP(&lock, &key);
    c)(cs)
    d) lock = f;
```



P1 : a; -- SWAP --- Pre
P2 : a; SWAP
P3 : a; SWAP

(1) Always Guarantee Mutual exclusion.
(2) It always guarantee Progress
(3) But it fails to guarantee Bounded wait

5. The enter_cs() and leave_cs() functions to implement Critical section of o. 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;
}
```

In the above solution, x is a memory location associated with 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 ~~more~~ Processes enter CS in FIFO order
IV. More then one process can enter CS same time

Which of the following statement are TRUE

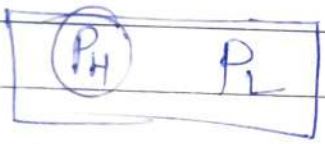
(A) I only (B) I & II only
(C) II and III (D) IV only.

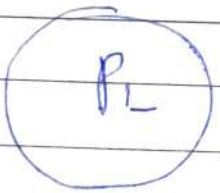
Priority-Inversion Problem:

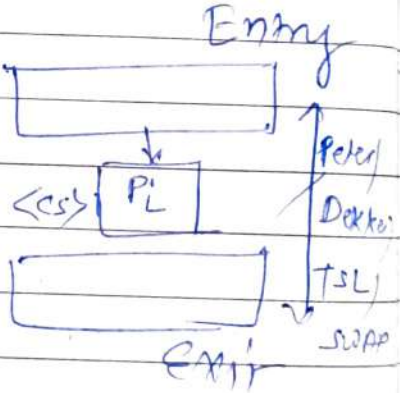
<All Correct Busy wait solution suffers from Priority inversion Problem>

: PreEmptive Priority based Scheduling

H > L







t:

"Deadlock"

→ If P_H is not interested to enter into CS then there is no problem.

→ If P_H is also interested to enter "CS"

Priority Inheritance

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)
entry {  while (Fetch_and_Add(L,1))
         L = 1;
      }
Exit: ReleaseLock(L)
      {
         L = 0;
      }
```

This implementation

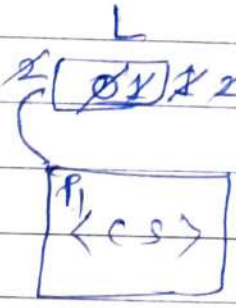
(A) fails as L can overflow.

(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.

Atomic
```
Fatch_and_Add (int x, int i)
{
    int rv;
    rv = x;
    x = x + i;
    return (rv);
}
```



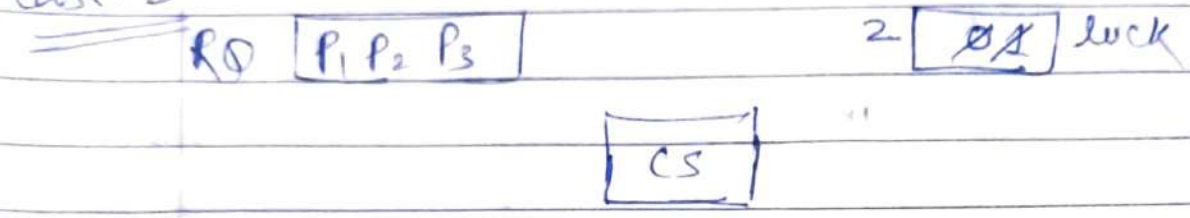
(P1) : FA : 0 int i (2B)

(P2) : FA : 1;
FA : 1

```
int lock = 0
Entry  while (Fatch_and_Add (&lock, 1))
           L = 1;
       <CS>
Exit   [L = 0]
```

```
int Fetch_And_Add(x, i)
{
    int rv;
    rv = x;
    x = x + i;
    return(rv);
}
```

Case - I



t_1 : P_1 : F_A_A : → 0
t_2 : P_2 : F_A_A : → 1

Case II

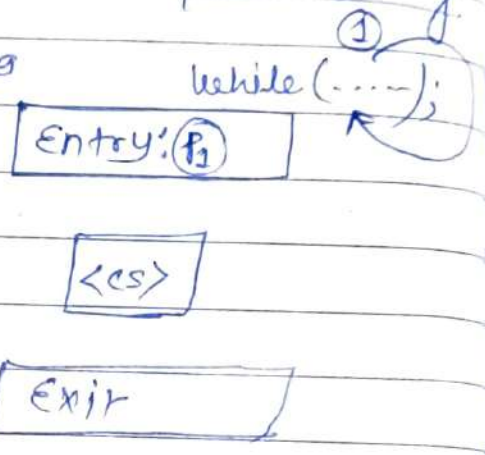
t_1 : P_1 : F_A_A : 0 : ⟨CS⟩
t_2 : P_2 : F_A_A : 1 ; Pre : L=1 (before setting L=1)
t_3 : P_1 : ⟨CS⟩ ;
t_4 : P_2 : L=1 FAA=1

Which of the following scheduling techniques probably suffer from starvation

a) FCFS X
b) SJF
c) Priority
d) Round Robin. X

II. Blocking Mechanisms / Non Busy waiting :-

TQ = 10 9



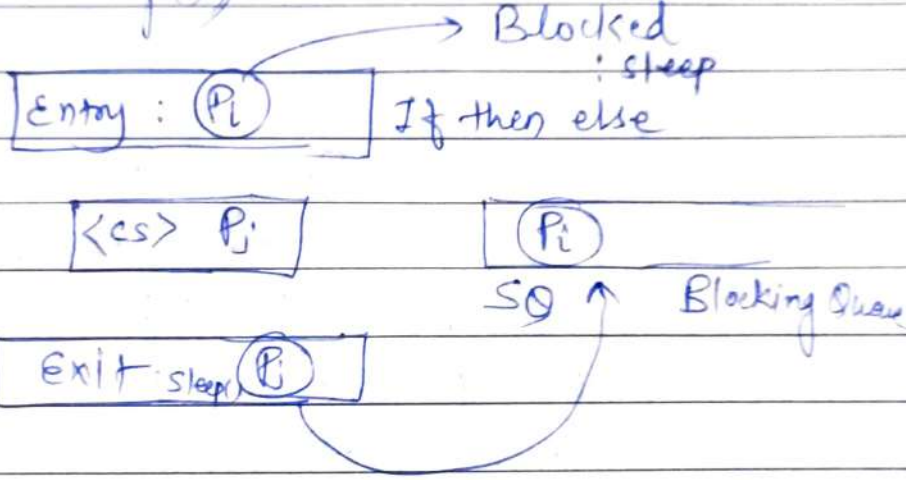
< Ps to ~~provide~~ avoid wastage of CPU Time, in the form of loops >

: if-then-else

: Sleep and Wakeup
: Semaphore
: Monitors

(1) Sleep() & wakeup()

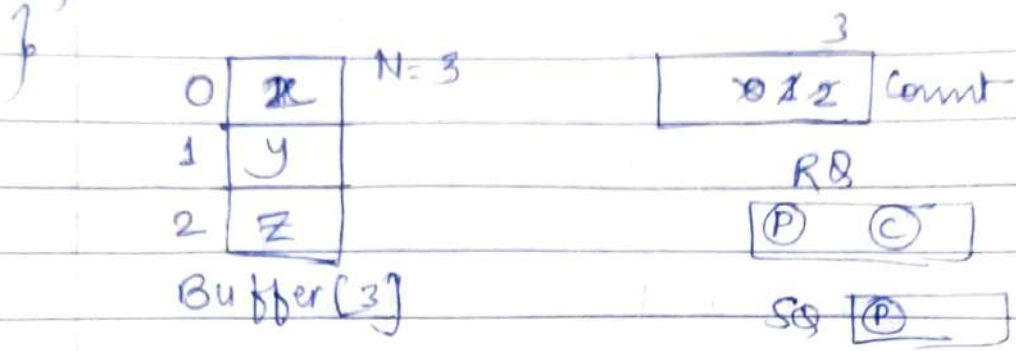
- → Blocking Construct
- → Multiprocess solution.
- → are Operating system primitives.
- → When process executes
  Sleep() :→ Gets blocked
  till some other process wakes it up
  (wakeup())



Producer Consumer Problem 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) Produce_item(item C)



t_1: C : L; C; J; Pre; <Sleep>

t_2: P : ..... Sleep();

t_3: C : Sleep();

Inconsistency

Deadlock

SEMAPHORES :-

→ Blocking Constructs
→ OS Based
→ semaphore is an operating System Resource (S/W)
→ General purpose utility.

{ – <CS>
  – Requirements of classical IPC Problem
  – Concurrency Mechanisms.
}

+ Practically In use tool (OS)

→ It was proposed & implemented by Edwin Dijkstra.

→ Semaphore is implemented as an Abstract Data Type.

→ Operations.
  - Atomic
    - KM
    - N.Pr.
  - Down() / wait() / P()
  - up() / signal() / V()

Definition :- Semaphore is a variable <ADT> : SEM> that variable takes only integer values;

Types

(BSEM) Binary (Mutex) → Mutual exclusion <0,1>

(CSEM) Counting (General) <−∞ to +∞>

Var(x)

Primitive <int, char, float>

Non Primitive user defined.

→ Binary semaphore is a subset of Counting semaphore

Counting semaphore
General semaphore <−∞ to +∞>

```
Typedef
    Struct
    {
      int values;
      QueueType L;
    }CSEM;
```

→ List of PCB's of those processes that get blocked, while performing

'DOWN' operation unsuccessfully.

UM

```
CSEM S;
S.value = 1;
(S = 1);

DOWN (S);
 <Si>
<Next_Statement>
```

K.M.

```
DOWN (CSEM S)
{
1. S Value = S Value - 1;
2. if (S.value < 0) //unsuccess
{
Put this Process
PCB in S.L. (Q)
& Block the Process (Sleep)
}
else
    return; //success
}
```

CSEM S;
S = 5

How many DOWN operations can be Carried out by Processes successfully?

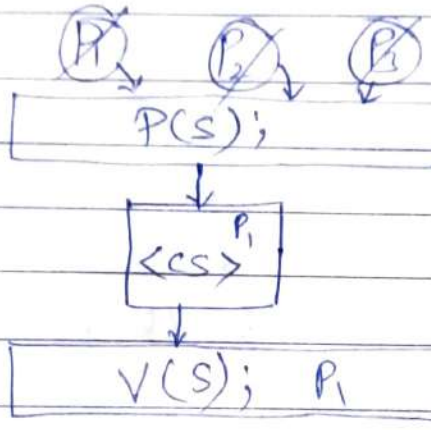
Soln:

S = ~~5 4 3 2 1 0~~ -1, -2

: 1, ~~1~~, 1, 1, 1

: 1, 1

1. positive value of Semaphore indicates that those many Down operations can be carried out successfully.
2. After carrying out a series of Down operation, if the value of Semaphore becomes -ve, then the magnitude -ve value indicates the no of Blocked processes;

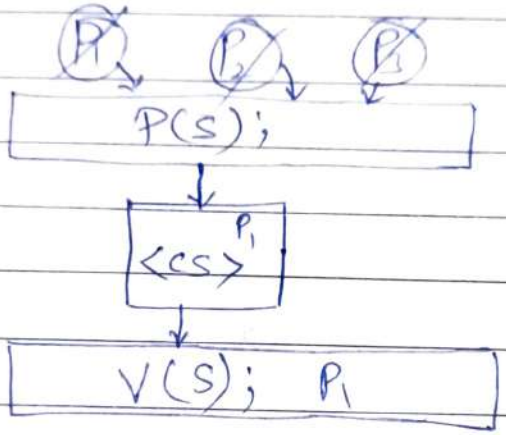


CSEM S;
S = 0
-1
-2
-3

CSEM
S = 3
2
S = 1;
0
-1
P2 P3
-1

up(SEM S)
{
1. S.value = S.value + 1;
2. if (S.value <= 0)
{
select a Process from
S.L.(Q) & wakeup();
}

1. positive value of semaphore indicates that those many down operations can be carried out successfully.
2. After carrying out a series of down operation, if the value of semaphore becomes -ve, then the magnitude -ve value indicates the no of Blocked processes;



CSEM S;
S = 0
-1
-2
-3

CSEM
S = 3

up(SEM S)
{
1. S.value = S.value + 1;
2. if (S.value <= 0)
{
select a Process from
S.L.(Q) & wakeup( );
}
}

Q1) CSEM S: 8 ;

Operation : 10P ; 1V ; 15P ; 2V ; 6P ; 3V

S = ?

Solu: 8 # 10 + 1 − 15 + 2 − 6 + 3

$-2+1-15+2-6+3$

$-1-15+2-6+3$

$-16+2-6+3$

$-14-6+3$

$-20+3$

-17 L $P_1 P_2 P_3 \ldots P_{17}$

Q2) CSEM S = . ;

Operations : 12P ; 4V ; 6P ; 3V ; 8P ; 1V

$S = -6$

$-6 = x - 12 + 4 - 6 + 3 - 8 + 1$

$-6 = x - 8 - 6 + 3 - 8 + 1$

$-6 = x - 14 + 3 - 8 + 1$

$-6 = x - 11 - 8 + 1$

$-6 = x - 19 + 1$

$-6 = x - 18$

$x = 18 - 6$

$x = 12$

Q: Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operation and 12 V(S) operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _____.

x(initial) = ?

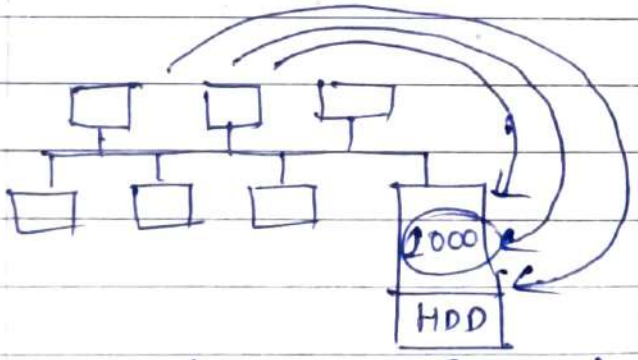
$$[x - 20 + 12] = -1$$

$$x - 8 = -1$$

$$x = -1 + 8$$

$$\boxed{x = 7}$$ Ans

Client Server Environment



<Data base> (Server)

Server can not manage above 1000 process and from clients Lacks of information (request) come

```
CSEM S=1000;
  P(S);
 <DB>
  V(S);
```

Semaphore of initial value application dependent होता है।

Binary Semaphore <0,1>

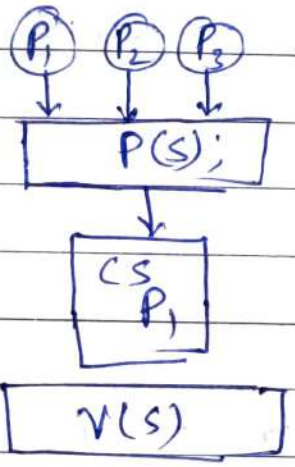
```
Typedef
        Struct
        {
                                int x;
        enum value(0,1);
        QueueType L;
        } BSEM;
```

→ list of PCB's of those processes that gets blocked while performing down operation unsuccessfully.

Um

```
BSEM S;
S=1;
Down(S);
< Next_statement >
```

```
DOWN (BSEM S)
{
  if (S value == 1)
  {
    S value = 0;
    return;
  }
  else
  {
    put this process (PCB) in SL(Q)
    & Block it (sleep());
  }
}
```



BSEM S=1;
S=1;
L 0
P2 P3

```
UP(BSEM S)
{
  if (S.L.(Q) is not Empty)
  {
    Select a Process from
    S.L.(Q) & wakeup();
  }
```

```
    else
    {
        S.value = 1;
    }
}
```

Cases of Binary semaphore :-

1. BSEM S;
   S = 1;
   P(S);
   S = 0
   status = Successful

2. BSEM S;
   S = 0;
   P(S);
   S = 0
   status = unsuccess.

3. BSEM S;
   S = 0
   V(S);
   S = 0 → 0 → if queue is not Empty
   S = 0 → 1 → if queue is Empty.
   Status = Success.

4. BSEM S;
S=1; <Implies CS is free & 'Q' is Empty.)
V(S):
S = 1
Status = Success.

5. BSEM S;
S=0 <Initial>
V(S); → first operation <'Q' is Empty)
S = 1
Status: Success

6. BSEM S=1;
Operations: 10P; 2V; 18P; 3V; 4P; 5V;
1) S = ?
2) Size of 'Q'[L] = ?

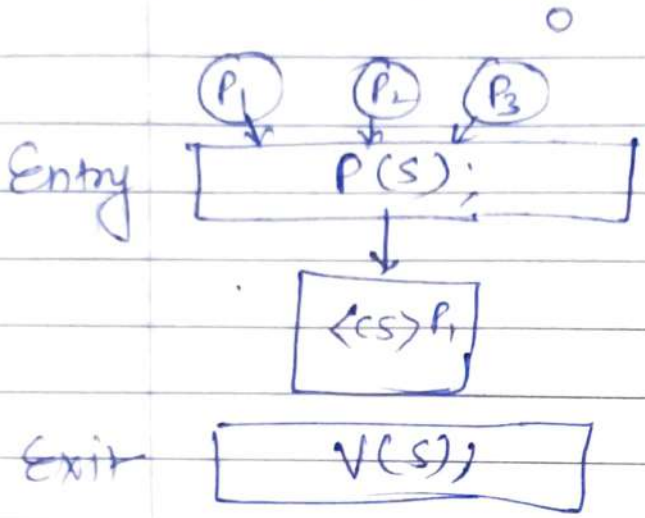
21 Process

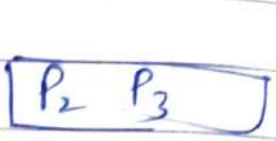
$-9+2-18+3-4+5$
$-31+10$
$=-21$

* As long as there is a process in "CS" & processes in Block 'Q' then the value of BSEM must be '0';

Both Binary and Counting Semaphores Can be used to Solve the problem of Critical section

BSEM S = 1;





1 : M.E ✓
2. Progress ✓
3. Bounded Wait

Q BSEM S = 1 T = 1

```
Pr(i)
{
  while (1)
  {
    P(S);
    P(T);
    <CS>
    V(T);
    V(S);
  }
}
```

```
Pr(j)
{
  while(1)
  {
    P(T);
    P(S);
    <CS>
    V(S);
    V(T);
  }
}
```

Does is it guarantee ME? How about Deadlock?

Ans It gnarantee Mutual Exclusion but is not Deadlock free.

if deadlock not happened then Pr(j) Sequence should be changed.

Q
```
BSEM S=1, T=0
Pr(i)                          Pr(j)
{                              {
  while(1)                       while(1)
  {                              {
   P(T);                          P(S);
  Print('1');                    Print('0');
  Print('1');                    Print('0');
  V(S);                          V(T);
  }                              }
}                              }
```

Q what is the output of program

Ans 0011001100 11- - -

Does this guarantee is Mutual Exclusion

* Application of SEMAPHORE.

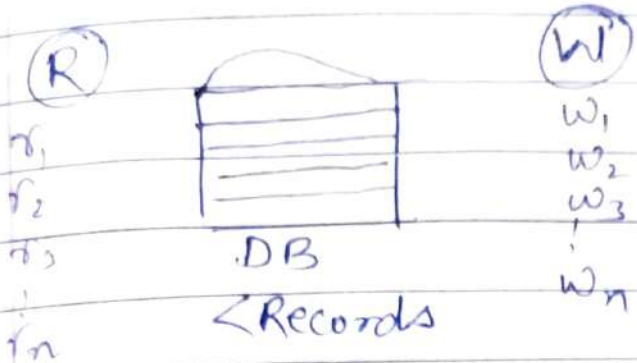
II Classical IPC Problem
- Producer Consumer
- Reader writer Problem
- Dining-Philosopher.

1) Producer Consumer :-

```
#define N 100
int Buffer[N]
CSEM Empty = N
   <No. of Empty slots>
CSEM full=0;
  <No. of full slots> <No. of Data items>
BSEM mutex=1;
   <used b/w (P) & (C) to ensure mutual
    exclusion on buffer>
Void Producer(void)
{ int itemp, in=0;
  while(1)
  {
   a) itemp = Produceritem();
   b) Down(Empty);      } Entry section
   c) DOWN (Mutex);
   d) Buffer[in] = itemp;
   e) in = (in+1)%N;
   f) up(mutex);
```

```
        g) up (full);
    }
}

void consumer (void)
{
    int itemC, out = 0;
    while (1)
    {
        a) DOWN (full);
        b) DOWN (mutex);
        c) itemC = Buffer [out];
        d) out = (out + 1) % N
        e) up (mutex);
        f) up (empty);
        g) Process_item (Item());
    }
}
```

Reader - writer Problem



There is no Empty database
There is no full Condition in database
1) Database is Never be full
2) Database is never be empty.

```
BSEM  mutex = 1
{
    P(mutex)
      <DB>
    V(mutex)
}
```

If Single Reader Writer is possible then above solution is Satisfied But here multiple Readers and writers access the database.

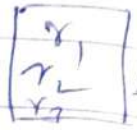
First R-W Problem: Starvation to Writer

t_1 : r_1 & w_1 : Any one

t_2 : r_1 ✓

t_3 : r_2 ✓

t_4 : r_3 ✓



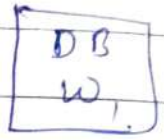
t_1 : r_1 and w_1 : Anyone

t_2 : w_1 ✓ ; r_1 wait

t_3 : w_2 × ; wait

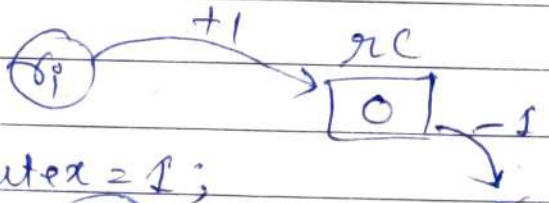
t_4 : w_3 × ; wait

t_5 : r_2 × ; wait



Implementation of first R-W using Semaphore

```
int rc = 0; // Reader's Count //
```



BSEM mutex = 1;
<used by (R's) to update rc>

BSEM db = 1
<used by (R's) & (W's) access <DB> as <CS>>

<1'st reader is the leader of all other Readers>

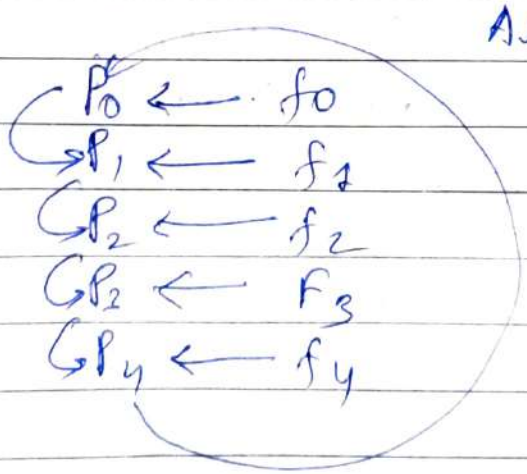
```
void writer(void)
{
  while(1)
  {
    a) DOWN(db);
    b) <DB-Write>
    c) UP(db);
  }
}

void Readers(void)
{
  while(1)
  {
    a) DOWN(mutex);
    b) rc = rc+1;
    c) if(rc==1) Down(db);
    d) up(mutex);
    e) <DB-READ>
    f) down(mutex);
    g) rc = rc-1;
    h) if(rc==0) up(db);
    i) up(mutex);
  }
}
```

DINING PHILOSOPHER'S PROBLEM

'N' Philosophers ($N \geq 2$)
Spaghetti (Noodles)

```
#define N 5
Void philosopher (int i)
{
  while(1)
  {
    a) Think(i);
Preemption → b) take_fork(i);
    c) take_fork((i+1)% N);
    d) eat
    e) Put_fork(i)
    f) Put_fork(i+1) % N
  }
}
```

All → hungary



Q. Without Deadlock, for N=5, what is the max # of Philosophers that can be eating ——

P_0 : f_0 f_1 ✓

P_1 : X

P_2 : f_2 f_3 ✓ (Max Concurrency)

P_3 : f_3 X

P_4 : f_4, f_0 X

N = 6

P_0 : f_0 f_1 ✓

P_1 : X

P_2 : f_2 f_3 ✓

P_3 : f_3

P_4 : f_4; f_5 : ✓

P_5 : f_5 X

How to prevent/avoid Deadlock in Dining philosophers.

(Semaphore based)

→ Put Semaphore Control on taking and releasing the forks

Non Semaphore based

Deadlock free operation

Out of N Philosopher

Let (N-1) ⟶ L → R

& 1 ⟶ R → L

→ Let even no. philosopher : L – R

→ Odd No. of Phijosopher : R – L