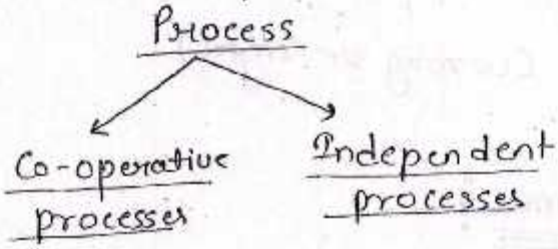
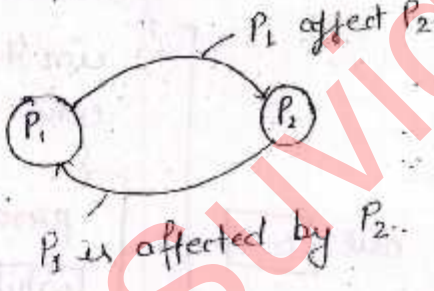
SYNCHRONIZATION

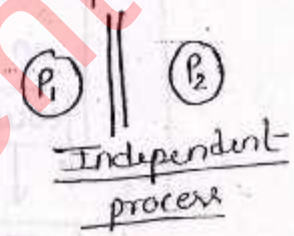
"The process are categorised on the basis of their synchronization are of two types"-



The execution of one process affects or affected by other process then those processes are said to be cooperative processes. otherwise we have said to be independent processes.



These two processes are co-operative type.

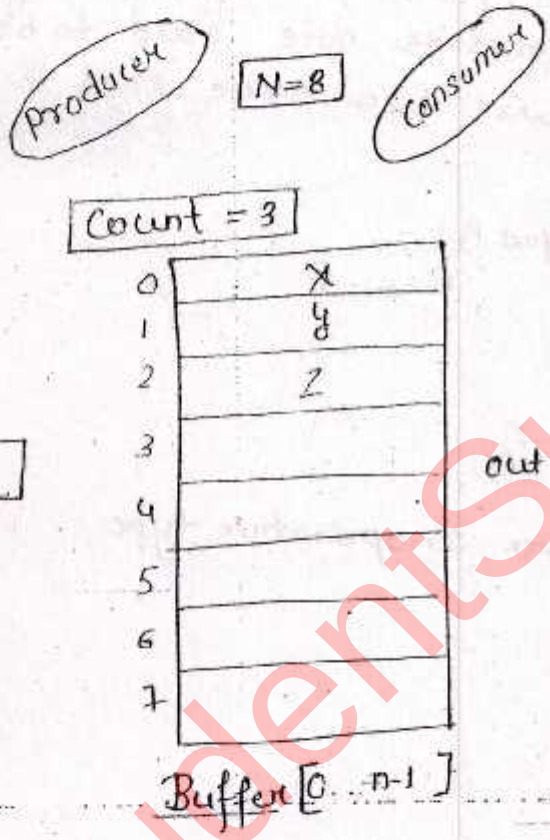


If we want to implement synchronization b/w processes, processes must be co-operative processes.

Three steps:

1) The problem arises NOT having the synchronization b/w the processes
2) The conditions to be followed to achieve synchronization
3) The solutions (wrong or right)

1) Producer - Consumer:



Producer Code

```
int count = 0;
void producer (void)
{
    int Itemp;
    while (TRUE)
    {
        produce-item (Itemp);
        while (count == N);
        Buffer [in] = Itemp;
        in = (in+1) mod N;
        count = count + 1;
    }
}
```

I. load Rp, m [count]
II. INCR Rp
III. Stole m[count], Rp

There is a common shared buffer, producer produce an item & placed in buffer, from the same buffer, consumer consumes the item from buffer.

→ in - is a variable used by producer to identify the next empty slot in the buffer.

Consumer code

```
void consumer(void)
{
    int ItemC;
    while (TRUE)
    {
        while (count == 0);
        ItemC = Buffer[out];
        out = (count+1) mod N;
        count = count + -1;
        process_item(ItemC);
    }
}
```

I. load Rc, m[count]
II. DECR Rc
III. Store m[count], Rc

→ out is a variable used by consumer to identify from where it has to consume the item.

producer should know about the free available space, where the item will be placed in buffer.

→ count is a variable used by the producer & the consumer to identify no. of items present in the buffer at any point of time

Shared Resources

1. count variable
2. Buffer

Two conditions -

1) When the buffer is full, the producer is not allowed to produce the item into the buffer.

2) When the buffer is empty, the consumer is not allowed to consume the item from the buffer.

count = 0 (buffer is empty)

Itemp (item will produce) by producer

while (True) - infinite condition only - we want that producer keeps always checking for buffer is empty or not.

(while + ;) - loop executed, when condition is false.

(count == N) - buffer is full

in = (in + 1) mod 8

mod is used to repeat value from 0 to 7 (buffer)

eg in = 7, (7+1) mod 8 - it will go again to 0.

Rp - register used by producer

Rc - register used by consumer

Universal Assumption :-

"While executing the instruction, if the interrupt will comes, the interrupt will be served only after completion of current instruction."

Analysis of Producer-Consumer Code

1) Inconsistency Problem

Initially count = 3 ✓
in = 3
out = 0

P - producer - produce item
in = (in+1) mod N
= (3+1) mod 8
= 4 mod 8
in = 4

P → count → I $R_p \leftarrow m(count)$ R_p 3
II INC R_p R_p 4

→ interrupt occur,

C - Consumer - consume item
out = (out+1) mod N
= (0+1) mod 8
out = 1

C → count → I $R_c \leftarrow m(count)$ R_c 3
II Dec R_c R_c 2
III $m[count] \leftarrow R_c$
Count = 2 ✓

---

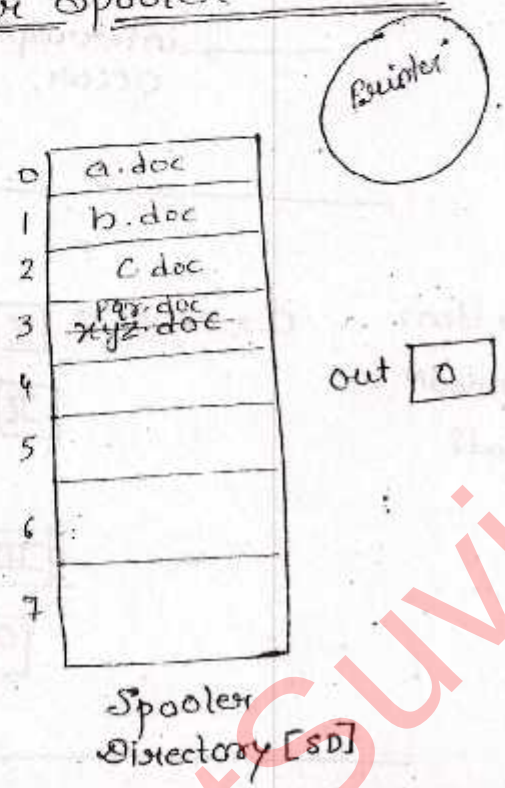
producer

P → count → III $m[count] \leftarrow R_p$
Count = 4 ✓

Inconsisteny arrived here

⇒ The producer and consumer are not properly synchronized while sharing the common variable counter. Hence it is leading to INCONSISTENCY while updating the counter variable.

2) Printer Spooler Daemon



Enter-file

I. load Ri, m[in]
II. Store SD[Ri], "F-N"
III. INCR Ri
IV. Store m[in], Ri

If there are 10 system having only one printer & from all 10 system, there is a print command at the same point of time, then printer will print one-by-one but first it is stored on spooler & keep on printing one-by-one later, from spooler directly directory.

in – is a variable used by all the processes to identify the next empty place in the spooler directory.

out – is a variable used by only printer to identify from where it has to print the document.

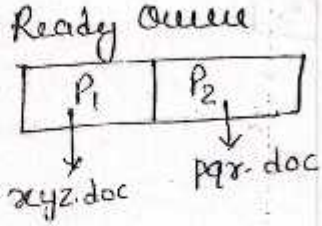
Shared Resources

1) I:N variable
2) Spooler directory

Ri → Respective process register
(Every process will have its own register)

F-N → File Name

Analysis



→ P_1 executing I_1 – load value of in ie. [in=3] in Ri [3]
→ P_1 executing II_2 – store file name in SD[3]
→ P_1 executing III – Ri incremented [4]

process P_1 is interrupted here & pre-empt to process P_2.

→ P_2 executing I – load value of in ie. [in=3] in Ri [3]
→ P_2 executing II – store file name in SD[3] (overwrite is happen)
→ P_2 executing III – Ri incremented [4]

↓
(loss of data) problem is occured

If all these 4 ins" are executed w/o pre-emption, the solution may be work fine but not guaranteed that pre-emption will not occur after II & III ins".

2) Loss of Data problem

⇒ The processes are not only properly synchronized while sharing the common variable 'in' Hence it is leading to **LOSS OF DATA**

3) Deadlock

⇒ If the processes are not properly synchronized while sharing the common resources then it is possible for **DEADLOCK**

Definitions

1) Critical section (CS):-

The portion of program text, where the shared variables or shared resources are placed.

Ex: `count = count+1` or `count = count-1` → Producer-Consumer Problem

2) Non-critical section (Non-CS):-

The portion of program text where the independent code of the processes will be placed.

Ex: `in = (in+1) mod N` or `out = (out+1) mod N` → Producer-Consumer problem

3) Race condition:

The final output of any variable depends on the execution sequence of the processes. This condition is called as race condition.

Ex: Analysis of Producer - Consumer Problem

P → I P → II C → I C → II C → III P → III	C → I C → II P → I P → II P → III C → III

II. Conditions to be followed to achieve Synchronization

(1) Mutual Exclusion (ME):

⇒ No two process may be simultaneously present inside the critical section at any point of time.

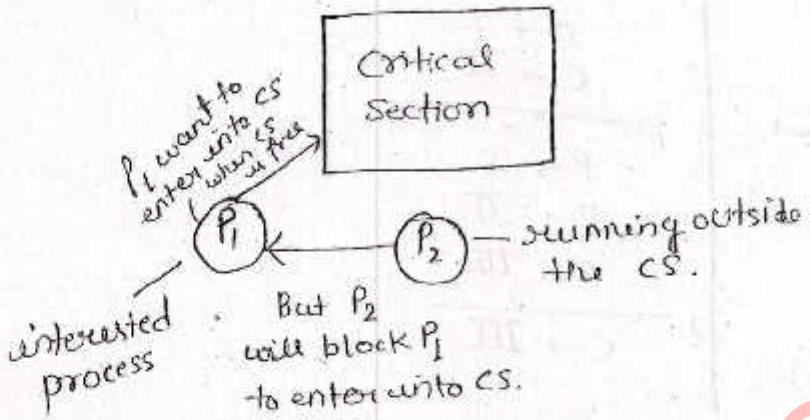
⇒ Only one process is allowed into critical section at any point of time.

Ex- in producer - consumer problem, count variable is shared variable, count = count + 1/-1 is critical section, both process - producer & consumer try to acess the critical section at same time & arrises the problem of inconsistency.

Both are present inside into CS.

2) Progress:

No process running outside the critical section should block the other interested process from entering into critical section when critical section is free.



3) Bounded Waiting:

⇒ No process should have to wait forever to enter into the critical section.

⇒ There should be a bound on getting chance to enter into CS.

⇒ If the Bounded waiting is not satisfied then it is possible for starvation.

4) No assumption related to H/w & the processor speed

III. SOLUTIONS:

I. Software Type

a) Lock variables
b) Strict alteration (or) Dekkers algorithm
c) Petersons algorithm

II. Hardware Type

a) TSL instruction set
   ↳ test & set lock

III. O.S Type

a) counting semaphore
b) Binary semaphore

IV. programming language compiler support type

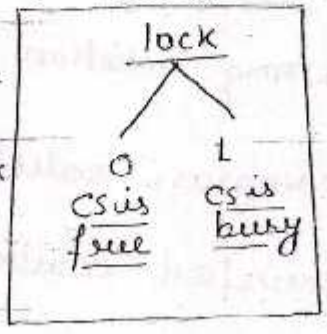
a) monitors

I. Software Type

(a) Lock Variables:

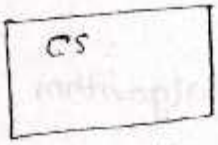
Entry-Section

I. load Ri, m[lock] — set Ri = 0/1 a/c to lock position (0/1)
II. CMP Ri, #0 ⇒ check cs is free/busy
III. JNZ to step I ⇒ if busy go to step I otherwise execute next statement
IV. Store m[lock], #1 ⇒ process enters into CS
V. C.S.
VI. Store m[lock], #0.
↳ this step means process is leaving the cs, no more in cs.
set lock = 0 (means cs is free)



Analysis:

Lock = 0 (CS is free)



$P_1 \rightarrow$ I

$P_2 \rightarrow$ II

$P_3 \rightarrow$ III

R_1 [0]

Interrupt occur

$P_2 \rightarrow$ I

$P_1 \rightarrow$ II

$P_3 \rightarrow$ III

$P_2 \rightarrow$ IV

$P_2 \rightarrow$ V

R_2 [0]

Interrupt

$P_2 \rightarrow$ IV

$P_1 \rightarrow$ V

Note:
We have proved that both the processes P_1 & P_2 are simultaneously present inside the Critical Section at the same time. Hence the mutual exclusion is not satisfied & so it is bound to be incorrect.

→ Wrong solution as to CPU

→ Progress satisfied.

→ Bounded waiting not satisfied.

(b) Strict alternation (or) Dekkes algorithm

(process takes turn to takes turn to enter into Critical section)

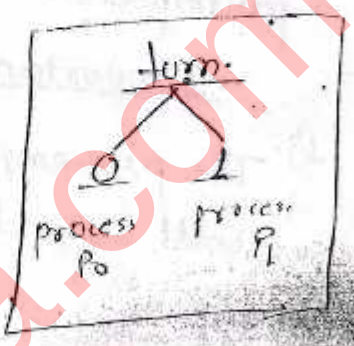
Process P_0 code

```
while(true)
{
    Non-CS();
    while(turn!=0);
    CS
    turn=1;
}
```

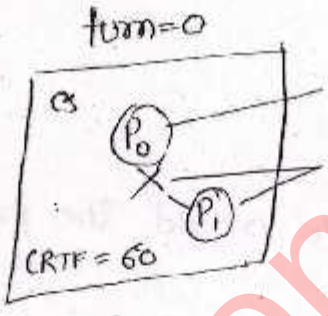
Remainder section

Process P_1 code

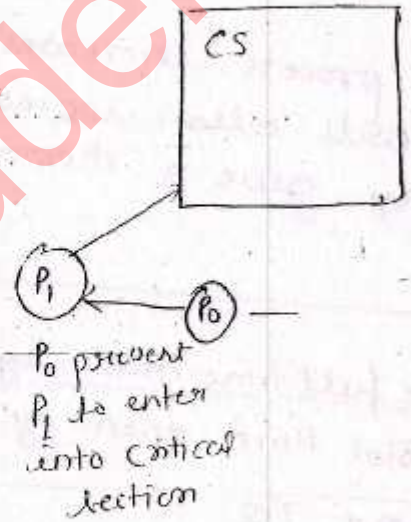
```
while(true)
{
    Non-CS();
    while(turn!=1);
    CS
    turn=0;
}
```



Analysis :-



When P_0 is in CS, P_1 doesn't & if P_1 is in CS then P_0 is not means mutual Exclusion is achieved.



1. ME is satisfied
2. progress is not satisfied.

Defⁿ of progress:
which process will go next into critical section is decided by only those process who want to enter go into critical section. In this decision the process in the reminder section and the process which take is not interested to go into CS should not paid participation.

Important Points

1) The pre-emption is just a temporary stop, the process will come back and continue the execution.

2) If there is any chance of solution becoming wrong by taking the pre-emption then only consider the pre-emption.

3) If any solution has deadlock the progress is not satisfied.

ageing

⇒ ageing is the concept generally used to avoid the problem of starvation.

⇒ If the waiting time of the process increases then the priority of the process will also increased and the process will defenetly give a chance to continue the execution.

CPU efficiency = $\frac{\text{Use full time made by the CPU}}{\text{Total time spent by the CPU}}$

$\frac{Q - TQ}{Q+T}$

T_{switch}

c) Peterson's Algorithm

(2-process solution)

```
#define N 2 (having two process)
#define TRUE 1
#define FALSE 2

int turn; (turn is a global variable, (which is used by the both process))
int interested[N]; — array of N value.
void Enter_Region (int process)
{
1. int other;
2. other = 1-process;
3. interested[process] = TRUE;
4. turn = process;
5. while(turn = process && interested[other] == TRUE);
}
```

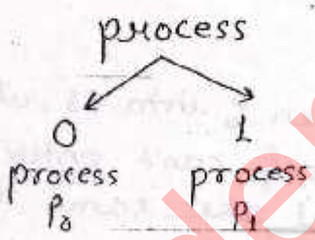
C.S

```
void Leave_Region (int process)
{
    interested[process] = FALSE;
}
```

Initially

interested[0] = FALSE

interested[1] = FALSE



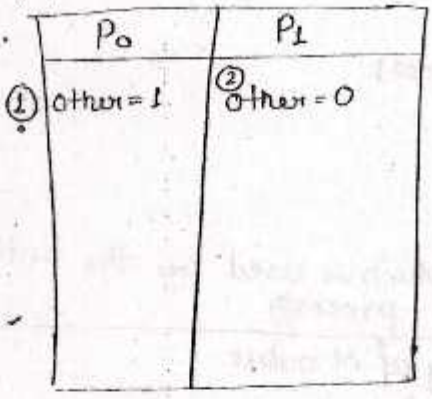
initially

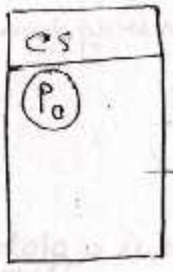
interested [0] = ~~FALSE~~ TRUE ①

interested [1] = FALSE ~~TRUE~~ ②

Analysis

turn = ~~0~~ ① 1 ②





① while (turn == process [&&] interested [other] == TRUE);

interested [i] == TRUE

0 0

true

False

false

Process P_0 enter into CS.

② while (turn == process [&&] interested [other] == TRUE);

interested [0] == TRUE

1 1

true

True

True

Process P_1 stop here & continue move into while loop.

Mutual Exclusion - satisfied. when P_0 into CS, at-then Process P_1 can't enter into CS at the same time

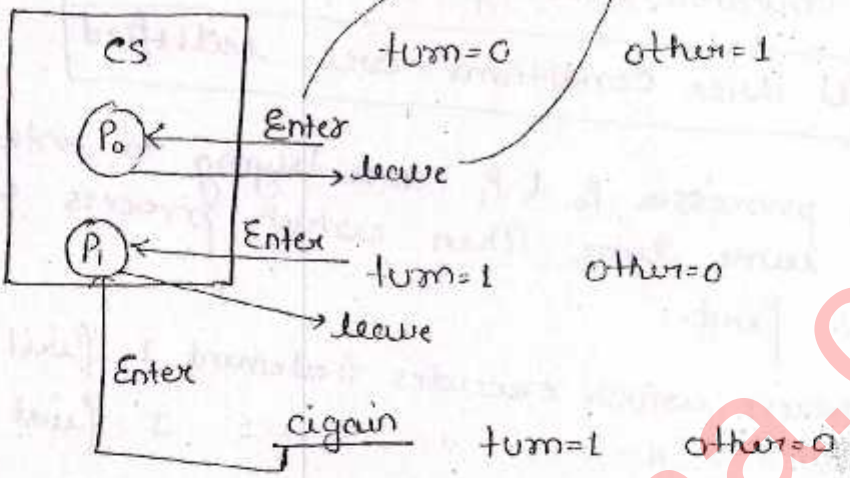
Note: If we are starting with Process P_1, then also when P_1 is in CS, P_0 can't enter into CS at the same time

Check it for progress condition

initially

interested[0] = false TRUE False.

interested[1] = false TRUE false TRUE



Progress is satisfied

- suppose when cs is empty P_0 want to enter into cs after turn = process = 0, P_0 is pre-empted. interested[0] = TRUE (P_0 not entered into cs, still cs is empty)
- then P_1 want to enter into cs. turn = 1, other = 0 interested[1] = TRUE

while (Turn == process && interested[0] == TRUE / false)

T T

True

P_1 want to enter into cs but can't

Note:- Here P_0 & P_1 both are interested process. but P_0 is in the remainder section.

Bounded waiting :

The no. of processes in any given "cs" is countable then the bounded waiting is said to be satisfied.

Bounded waiting is also satisfied

All three conditions are satisfied.

Q: Both the processes P_0 & P_1 are trying to enter into cs at the same time. Then which process will enter into cs first.

(a) The process which executes statement 2 first.
(b) " " " " " 3 first.
(c) " " " " " " "
(d) We cannot say.

Sol:- one will be satisfiable in all possible conditions.

Option (c) is correct

Case I

first P_0 execute statement 3 (first), P_1 after that P_0 (second)
first P_1 execute statement 4 (first), P_0 after that P_1 (second)

turn = 1, 0 interested[0] = ~~FALSE~~ TRUE
interested[1] = FALSE TRUE

while (Turn == process && interested [other] == TRUE);
interested [1] == TRUE

P_0 → 0 == 0
True
True
True (P_0 rotating into while loop) can't enter into CS.

P_1 → 0 == 1
False
interested [0] == TRUE
True
False (P_1 enter into CS).

Case II

when P_0 execute statement 3 (first), P_1 after that P_0.

→ P_0 is pre-empted

P_1 execute statement 4 (first)

while (turn == process && interested[other] == TRUE);

$1 == 1$ — True

interested[0] == TRUE — True

True (P_1 keep on going to check into while loop for false cond^n)

when P_0 execute stat^n 4 after P_1.

it make turn = 0

False ($0 == 1$) — True

False (P_1 enter into CS) first

II. Hardware Type

a) TSL instruction set:

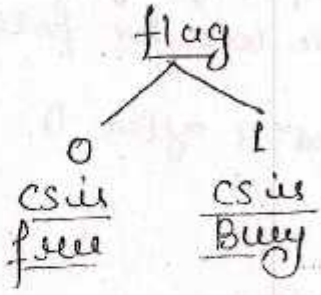
TSL- Test and Set Lock

TSL- Register flag :- copies the current value of flag into register and stores the value of '1' into flag in a single atomic cycle without any pre-emption.

means disable all the interrupts (this happens only at the h/w level).

Entry Section

I. TSL Ri, m[flag]
II. cmp Ri, #0
III. JNZ to step (I)
IV. C.S.
V. Store m[flag], #0



Analysis :-

Initially flag = 0 (nobody is present in CS) ... CS free

P_1 : I. $R_1 \leftarrow$ m[flag] R_1 0 & set flag = 1
II. cmp R_1, #0 True
III. JNZ to step (I) – Zero then execute next statement/ins" (not jump)
IV. CS. (enter) → flag = 1

P_0 : I. $R_0 \leftarrow$ m[flag] R_0 1
II. cmp R_0, #0 false
III. JNZ to step (I) – it will jump & again check whether CS is free or busy

P_0 is not allowed to enter into CS.

Mutual Exclusion is satisfied.

Check it for progress condition.

∴ flag = 0

Note:- If you say nobody is in CS mean CS is free then we have to set flag = 0.

P_1 wanted to enter into CS

P_1 : I. R_1 0

II. cmp R_1, #0 True.

III. not jump

IV. 

flag = 1

Unless or until the process will not execute the last statement process is in the CS.

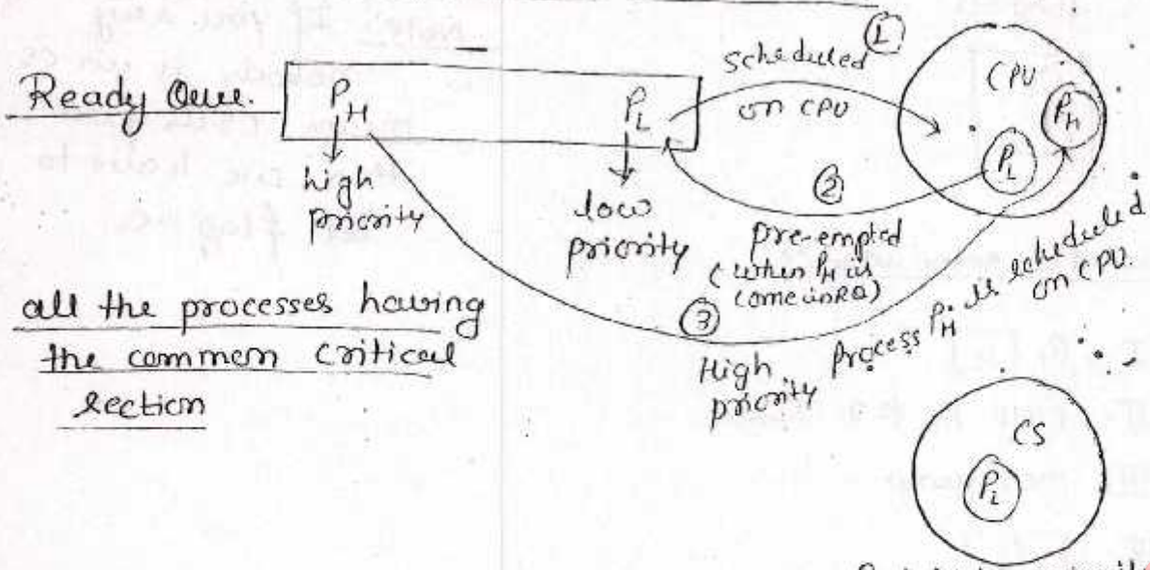
progress is satisfied

→ no. if process is not countable. so,

Bounded waiting is not satisfied

Soln.	M.E.	Progress.	Bounded waiting
1) Lock variable	NO	Yes	NO
2) Strict alternation (or) Dekkers algo	yes	NO	yes
3) Petersons algo	yes	yes	yes
4) TSL instruction set	yes	yes	NO

Priority Inversion Problem



all the processes having the common critical section

But high priority process P_H is not allowed to enter into CS. because it (CS) is already locked by P_L (low priority process).

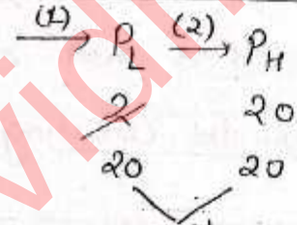
Problem ⟹ live lock

P_L → Ready

P_H → Run

The sol^n to above prob is → **Priority Inheritance**

(we don't want to pre-empt low priority process when it is in CS & & when high priority process will come) in RQ.



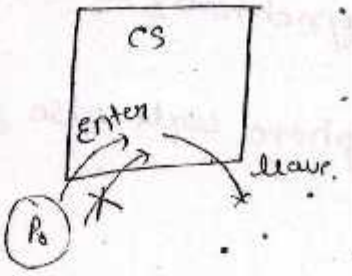
priority will be same then use FCFS algo.

(Inherit the priority of low priority process from low to high priority & make its priority of both process equal, then just execute the low priority process & later high priority process)

Strict alteration

P_0, P_1

progress is not satisfied

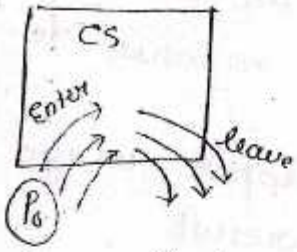


when 2nd time you try, you found that P_0 is not allowed to enter into CS bcz P_0 is not taking its own decision, it depend on decision of process P_1

Peterson algo.

P_0, P_1

progress is satisfied



Process P_0 does not depend on anyone to enter into CS (many no. of time it will be enter & leave the CS).

(here P_0 take it's own decision only).

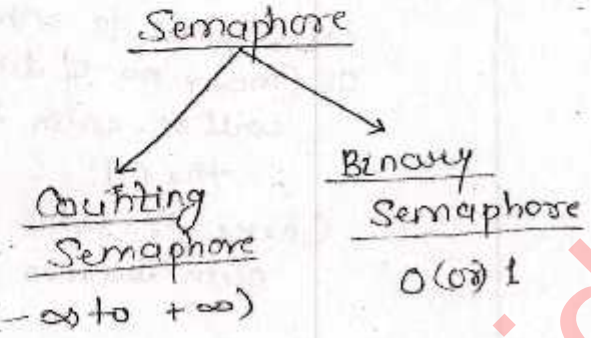
→ first check whether deadlock is there or not, if deadlock is not there then go further otherwise progress is not satisfied.

III. OS Type

Semaphore- It is an integer variable which is used by the various processes in a mutual Exclusive manner to achieve synchronization.

⇒ The improper usage of semaphore will also give improper result.

⇒



⇒ The two different operations are performed on the semaphore variable :-

1) Down() or wait() or P()

2) up() or signal() or V() or Release()

Note: Operations are same in both semaphore but way of performing is different.

Counting Semaphore :-

```
Down(Semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        Block the process and place its
        PCB in the suspended list();
    }
}
```

```
up (semaphore S)
{
    S.value = S.value + 1;
    if (S.value <= 0)
    {
        select a process suspended list()
        and wakeup();
    }
}
```

⇒ After performing the down operation, if the process is not getting suspended then it is called as succesful down() operation.

⇒ If it is succesful down() oper^n then only the process will continue the execution.

⇒ After performing the down operation, if the process is getting suspended then it is called as unsuccesful down() oper^n.

⇒ If it is unsuccesful down() oper^n then the process will not continue the execution.

⇒ $\boxed{S \geq 1}$ then only counting semaphore have succesful down() operation. (means only when initial value of semaphore is ≥ 1)

⇒ There is no unsuccesful up() oper^n. up() oper^n is always succesful.

⇒ The process performing up() oper^n will definitly continue the execution.

⇒ if S = +6 (6 successful down() operation will be performed).

⇒ if S = -6, (it means there is already 6 suspended process).

Que? Consider a system where a counting semaphore value is initialize to [+17]. The various semaphore operation like 23P, 18V, 16P, 14V, 1P are performed then what is the final value of semaphore?

(a) +7
(b) +8
(c) +9
(d) +10

Sol:

+17 - 23 = -6]
↳ -6 + 18 = +12]
↳ +12 - 16 = -4]
↳ -4 + 14 = +10]
↳ +10 - 1 = [+9] Ans.

Binary Semaphore:

```
Down (Semaphore S)
{
    if (S.value == 1)
        S.value = 0;
    else
    {
        Block the process and
        place its PCB in the
        suspended list ();
    }
}
```

```
up(semaphore S)
{
    if (suspended list() is Empty)
        S.value = 1;
    else
    {
        select a process from suspended
        list() & wakeup();
    }
}
```

⇒ After performing the down operation if the process is not getting suspended then it is called as successful down() oper^n.

⇒ If it is successful down operation then only the process will continue the execution.

⇒ After performing the down operation, if the process is getting suspended then it is called as unsuccessful down oper^n.

⇒ If it is unsuccessful down oper^n, then the process will not continue the execution.

⇒ The down() oper^n on the binary semaphore is successful only if the initial value of semaphore is 1.

⇒ There is no unsuccessful up() oper^n, the up() oper^n is always successful.

⇒ The process performing up() oper^n will definitely continue the execution.

Ques? Each process P_i, $i = 1$ to 9, executes the below code

```
repeat
    P(mutex);
    [C.S]
    V(mutex);
forever
```

(similar to while loop)

The process P_{10} executes the below code

```
repeat
    V(mutex);
    [C.S]
    V(mutex);
forever
```

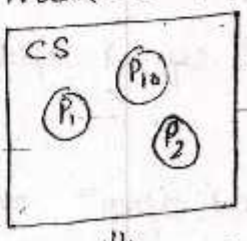
What is the max no of processes that may present inside critical section at any point of time? (The initial value of binary semaphore mutex = 1)

a) 2
b) 5
c) 9
d) 10 ✓

Sol:- Analysis

mutex = 1 ⇒ if P_2 want to enter, it can't but P_{10} is allowed

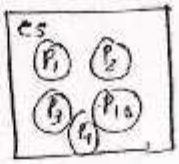
mutex = 0 ⇒ now P_2 is allowed



now P_3 is not allowed to enter

so, we have to remove P_{10}

means P_{10} is leaving the CS.



mutex = 0 ✗ now P_3 is allowed

0 ✗ again P_{10} is enter

0 P_{11} is enter into CS

when mutex = 0 (P_{10} is allowed to enter into CS, means leaving the CS)

& make mutex = 1 (so next process is allowed to enter)

when mutex = 0 (if P_{10} is present into CS, it has to leave the CS)

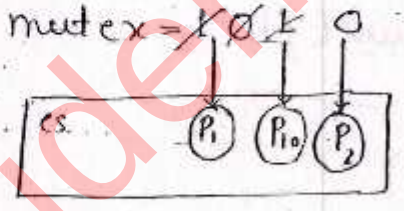
& make mutex = 1

Que? same as above.

Process P_{10} execute the below code.

```
repeat
    V(mutex);
    [CS]
    P(mutex);
forever
```

Sol.



max^m 3 process is in the CS at the same time

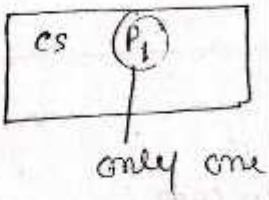
option (b) 3 Ans.

Qu: Same as it is until P_9

P_{10} execute the below code.

```
repeat (
    p(mutex);
    [C.S]
    V(mutex);
forever
```

Sol:



mutex = 1/0

only one process

Gate 2003 Qu. Consider the two concurrent processes executes this respective codes.

Process 'P' code

```
while (TRUE)
{
w: ————
print('0');
print('0');
x: ————
}
```

Process 'Q' code

```
while (TRUE)
{
y: ————
print('1');
print('1');
z: ————
}
```

what must be the binary semaphore operations on which w, x, y & z respectively and what must be the initial values of binary semaphore variables 'S' and 'T' inorder to get the print o/p always as 0011001100110.....

(a) $w = p(T)$, $x = v(T)$, $y = p(S)$, $z = v(S)$, $S = T = 1$;

(b) $w = p(T)$, $x = v(T)$, $y = p(S)$, $z = v(S)$, $S = 1, T = 0$;

(c) $w = p(T)$, $x = v(S)$, $y = p(S)$, $z = v(T)$, $S = T = 1$;

(d) $w = p(T)$, $x = v(S)$, $y = p(S)$, $z = v(T)$, $T = 1, S = 0$;

Sol:-

```
while (TRUE)
{
    P(T)
    print ('0')
    print ('0')
    V(S)
}
```

```
while (TRUE)
{
    P(S)
    print ('1')
    print ('1')
    V(T)
}
```

(a) P(S) mean P(1), $S = \not 0$ print '1' '1' first
o/p start with '11' incorrect

(b) P(S) same print '11' first incorrect

(c) " incorret

(d) P(T) $\Rightarrow$ $T = \not 0$

print 00

$S = \not 0$ 1

$S = \not 0$

print 11

$T = \not 0$ 1

Ques. Remaining part is same as above ques.
Which of the following will ensure that the o/p string never contains a substring of the form 01^n0 (or) 10^n1 where 'n' is odd?

(a) $\omega = P(S)$, $X = V(S)$, $Y = P(T)$, $Z = V(T)$, $S = T = 1$;

(b) $\omega = P(S)$, $X = V(T)$, $Y = P(T)$, $Z = V(S)$, $S = T = 1$;

(c) $\omega = P(S)$, $X = V(S)$, $Y = P(S)$, $Z = V(S)$, $S = 1$

(d) $\omega = V(S)$, $X = V(S)$, $Y = P(S)$, $Z = P(T)$, $S = T = 1$;

Sol: (a) P(S) P(T)

S=1 T=1 incorrect

o/p can't be predicted, it may be anything

it produce 0101

odd

(b) 0101 is also possible incorrect

(c): P_1 P_2

P(S) P(S)

0 1

preempt 0 1

V(S) V(S)

S=1

when P(S)

$S \neq 0$ print (00) or (11)

↓

can't $S = 1$ again print (11) or (00)

Before preemption- P_1 print (0) $S \neq 0$

after that, but P_2 can't print (1) bcz $S = 0$

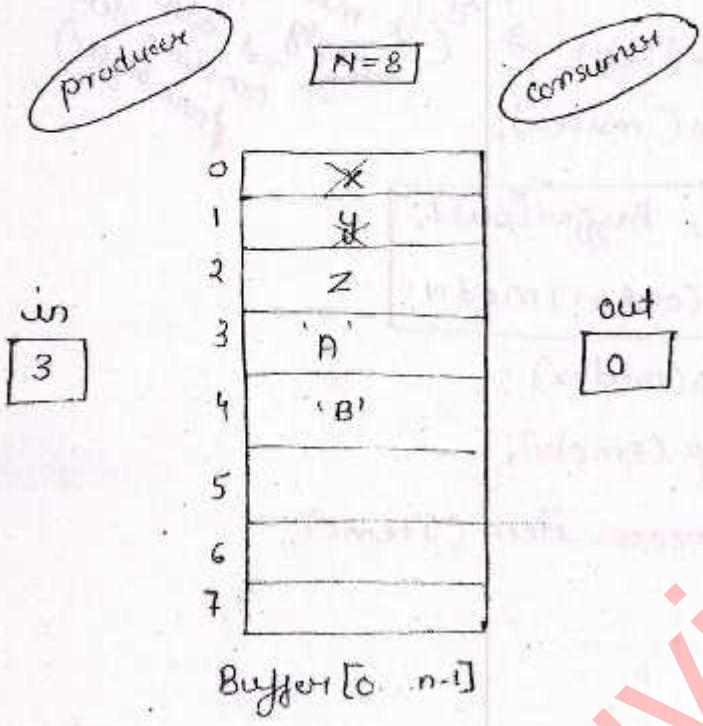
it produce either '00' or '11', there is no possibility of altering the o/p. it can't produce '0101'.

(d) it is also produce string containing 010 or 101 as a substring incorrect

Classical Problems of IPC

IPC (Inter Process Communication)

(1) Producer – Consumer with Semaphore




```
Semaphore mutex=1;
Semaphore Empty= N;
Semaphore Full=0;
void producer(void)
{
    int Itemp;
    while(TRUE)
    {
        produce_item(Itemp);
        down(Empty);
        down(mutex);
        Buffer[in] = Itemp;
        in = (in+1) mod N;
        up(mutex);
        up(full);
    }
}
```

Empty=0, full=8
(it means buffer is full, producer is not allowed to produce an item)
if producer want to produce, perform down(Empty) & get suspended.

```
void consumer (void)
{
    int ItemC;
    while (TRUE)
    {
        down (FULL);   ⇒ if full = 0 (it means buffer is empty, now consumer is not allowed to consume item from buffer)
        down (mutex);
        ItemC = Buffer[out];
        out = (out + 1) mod N;
        up (mutex);
        up (empty);
        process_item (ItemC);
    };
}
```

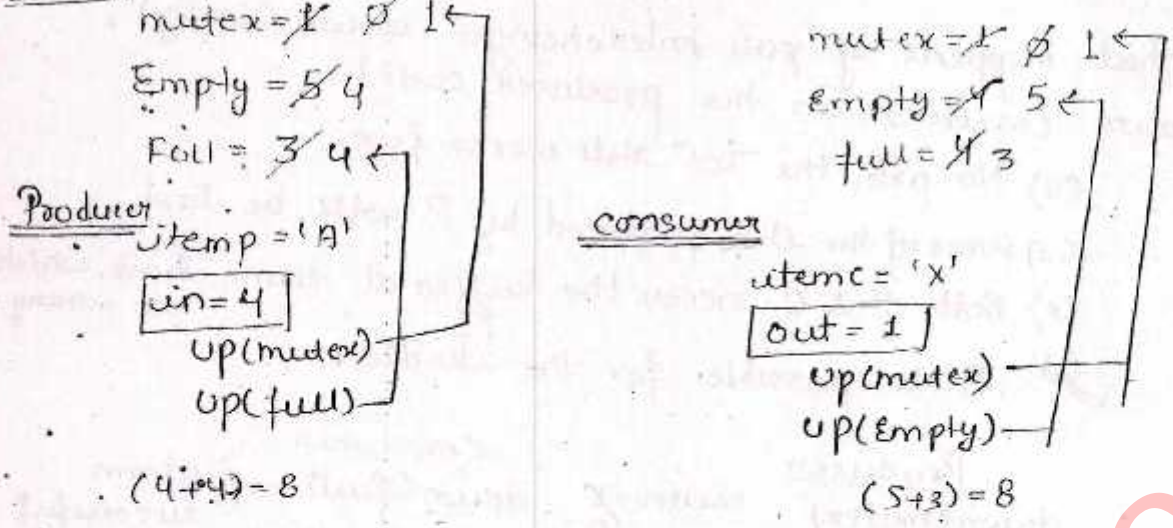
⇒ mutex - is a binary semaphore used by the producer and consumer to access the buffer in a mutual exclusive manner.

⇒ Empty - is a counting semaphore variable represents the no. of empty slots in the buffer at any point of time.

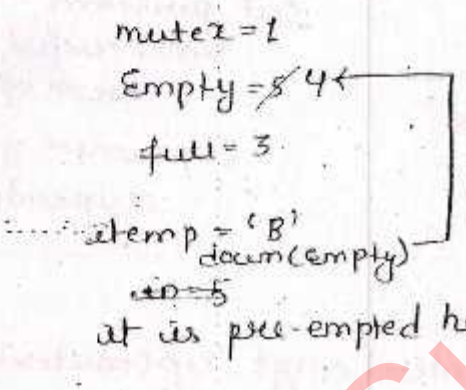
⇒ Full - is a counting semaphore variable represents the no of filled slot in the buffer at any point of time

Analysis :-

Case I:

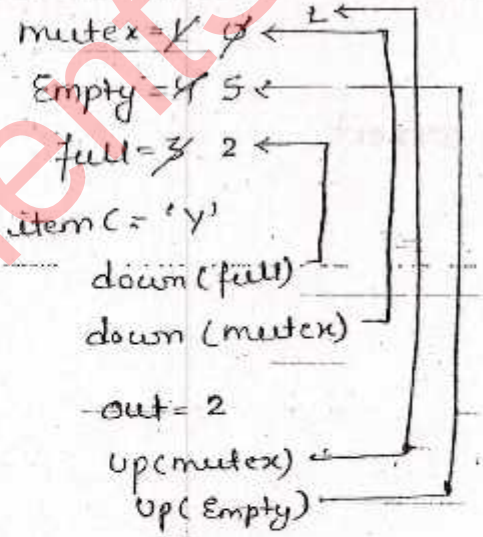


Case II: produce one more item

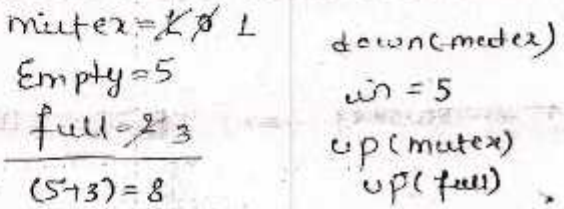


Empty + Full = N always.

consumer came to consume one item



producer came to reexecute the item execution



Que? What happens if you interchange down(Empty), down(mutex) in the producer code?

(a) No prob. the Sol^{n} still works fine.

(b) Some of the item produced by P will be lost.

(c) Both P & C access the buffer at same time, which is wrong

(d) It is possible for the deadlock

Sol:

Producer

down(mutex) mutex=$\not{0}$

down(Empty)

if Empty=0 (means buffer full)

(unsuccessful) Empty=$\not{0}$, -1 down oper^{n}.

producer get suspended

Consumer

down(full) — perform successful down oper^{n}.

down(mutex)

but mutex=$\not{1}$, 0

can't perform (unsuccessful down oper^{n})

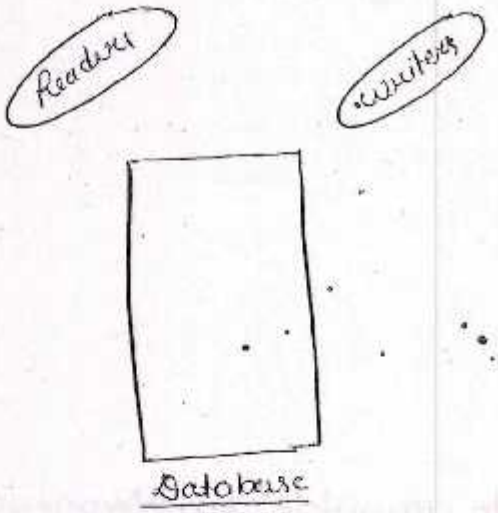
consumer get suspended.

Que? what happens if you interchange up(mutex) & up(full) in the producer code?

same options.

Sol: option (a) is correct

Readers – Writers Problem




```
int rc=0;
Semaphore mutex=1;
Semaphore db=1;
void Reader (void)
{
  while(TRUE)
  {
    down(mutex);
    rc = rc+1;
    if (rc ==1) down(db);
    up(mutex);
    D.B
    down(mutex);
    rc = rc-1;
    if (rc == 0) up(db);
    up(mutex);
    process_item();
  }
}
```

```
void writer(void)
{
    while (TRUE)
    {
        down(db);
        D.B
        up(db);
    }
}
```

4 conditions to be followed to provide synchronization b/w readers & writers :-

1. R→W (not allowed writer into database when it is already occupied by reader) ✗
2. R→R (when one reader is already access the database, another reader is allowed) ✓
3. W→R (when writer accer the database, reader is not allowed) ✗
4. W→W (when writer accer the database, another writer is not allowed) ✗

⇒ r_c is an integer variable, which represent the readers count ie. the no. of reader present in the database at any point of time.

⇒ mutex is the binary semaphore used by the readers in a mutual exclusive manner.

⇒ db is a binary semaphore variable used by the readers and writers in a mutual exclusive manner.

Analysis

mutex = 1 → 0 → 1
db = 1 → 0 → 1
rc = 0 → 1 → 0

⇒ Reader

down(mutex)
rc = rc + 1
rc == 1 down(db)
up(mutex)

D.B
[R1]

down(mutex)
rc = rc - 1
rc == 0 up(db)
up(mutex)

DB

∴ mutex = 1
db = 1 → 0 → 1
rc = 0

writer

down(db)

[W]

up(db)

R → W ✗

→ if writer want to enter into DB, when reader is already in the DB

mutex = 1
db = 0
rc = 1

writer

down(db) db = 0, -1
unsuccessful
down operⁿ
(suspended)

R → R ✓

→ if reader want to enter into DB, when reader is already in the DB

mutex = 1 → 0 → 1
db = 0
rc = 1 → 2

Reader

down(mutex)
rc = rc + 1
up(mutex)

[R1 R2]
DB

Allowed

⇒ writer

mutex = 1 → 0
db = 1 → 0
rc = 0 → 1

DB
[W1]

W → R ✗

→ if reader want to enter into DB, when writer is already in the DB.

rc == 1 → down(db)
db = 0, -1
(get suspended)
unsuccessful down operⁿ

W → W ✗

if one writer is already into DB, another writer want to enter.

db = ϕ_1 − 1 ←
down(db)

(another writer get suspended)
unsuccessful down operⁿ.

Que.1 What happen if you interchange down(mutex) & rc = rc+1 in the reader's code?

option

(a) No problem, the solⁿ still works fine.

(b) multiple Readers are not allowed

(c) Readers and writers both can access the DB at the same time

(d) None of the above.

mutex = 1, ϕ, 1
db = 1, 0
rc = ϕ, 1, 2

Reader

```
while (TRUE)
{
rc = rc+1
down(mutex)
if (rc == 1) down(db)
up(mutex)
DB
```

R1 R2 both preempted

Writer

```
down(db)
DB
```

← successful down operⁿ

option (c) is correct

Gate 2011

```
int R=0, W=0;
Semaphore mutex=1;
void Reader (void)
{
  L1: down(mutex);
    if (W==1)
    {
      [up(mutex)] ---> (1)
      goto L1;
    }
    else
    {
      R = R+1;
      [up(mutex)] ---> (2)
    }
  [D.B]
  down(mutex);
    R = R-1;
  up(mutex);
}
```

```
void writer (void)
{
  L2: down(mutex);
  if ( [W==1 (or) R≥1] ) ---> (3)
  {
    up(mutex)
    goto L2;
  }
  W=1;
  up(mutex);
  [D.B.]
  down(mutex);
  W=0;
  up(mutex);
}
```

What should be the values of blanks 1, 2, 3 respectively in order to synchronize the classical readers & writers?

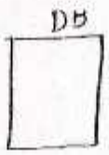
(a) up(mutex), down(mutex), W==1

(b) down(mutex), up(mutex), W==1

(c) down(mutex), up(mutex), W==1 (or) R≥1

(d) ~~down~~ up (mutex), up(mutex), W==1 (or) R≥1

Sol: (a)



(DB is empty)

mutex = 1, 0, -1
R = 0, 1
W = 0

R = R+1

down(mutex)

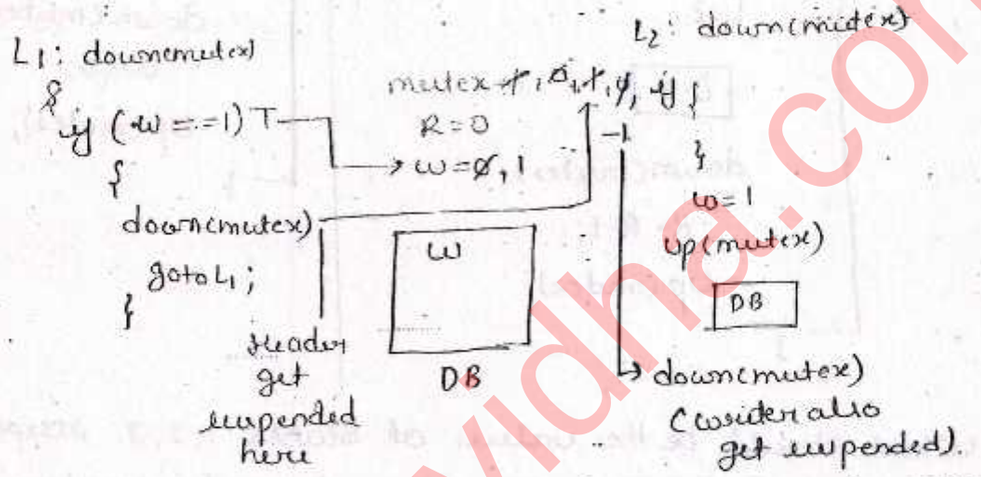
(Reader get suspended) but it is not the cond^{n}. When DB is empty, reader is allowed to enter.

(a) is incorrect

(b) if writer is trying to enter into DB (no writer or no reader is present in the DB) then only writer is allowed. here only (W == 1) ✓ (R ≥ 1) ✗ — it not mentioned

(b) is incorrect

(c)



(c) is incorrect

(d) Correct

Qu.? What happens if you interchange [w=1], [up(mutex)] in writer's code?

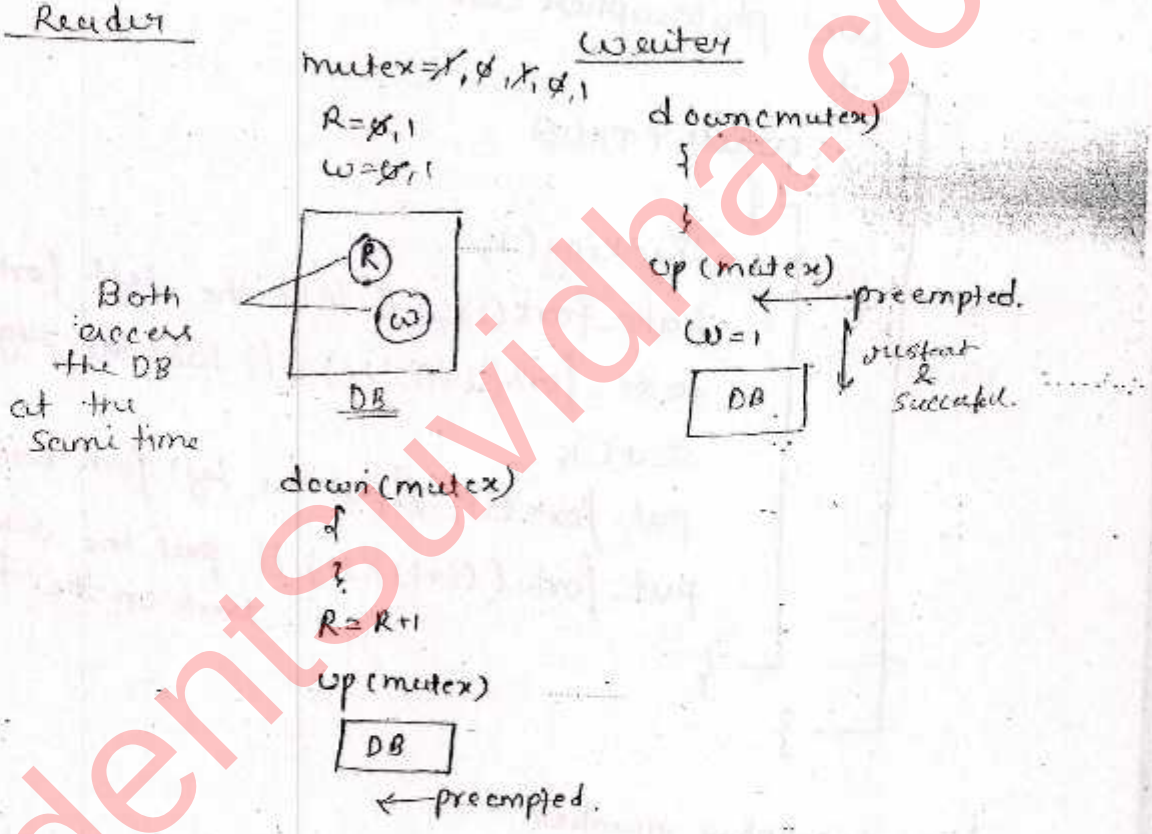
(a) No problem, the sol^n will works fine.

(b) Multiple readers are not allowed which is wrong

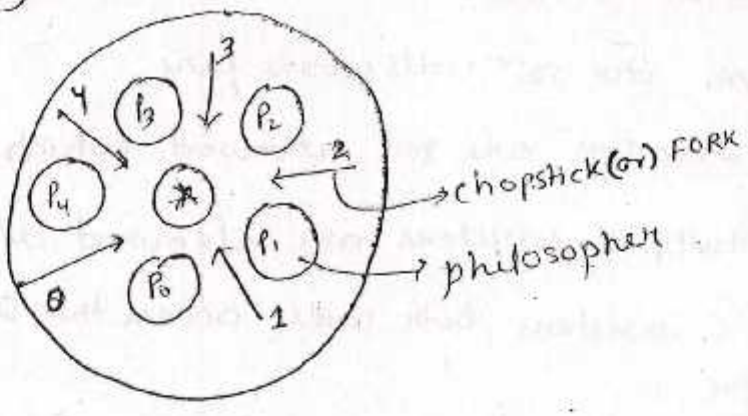
(c) Only multiple writers are allowed into D.B.

(d) Readers & writers both will access the DB at the same time.

Sol.



Dining Philosophers




```
void philosopher (int i)
{
    while (TRUE)
    {
        Thinking();
        take_fork(i);          // take the left fork
        take_fork((i+1)%N);    // take the right fork
        eat();
        put_fork(i);           // put the left fork back on a table
        put_fork((i+1)%N)      // put the right fork back on the table
    }
}
```

⇒ i → philosopher number

⇒ initially philosophers are in thinking state.

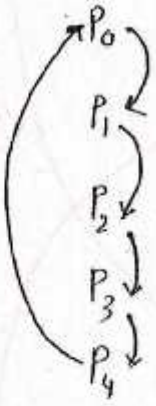
⇒ when they feel hungry then go in eating state.

for process P_4 → left fork $(i=4) = 4$

→ right fork $(4+1)\%N \Rightarrow (4+1)\%5 \Rightarrow 0$

Problem - when all philosopher are get hungry, all will pick up the left fork first & when they try to attempt to take the right fork, at the same tym the right fork is not available.

Then all the philosophers will wait on each other for the right fork & they are moving to d will going to deadlock.



Deadlock

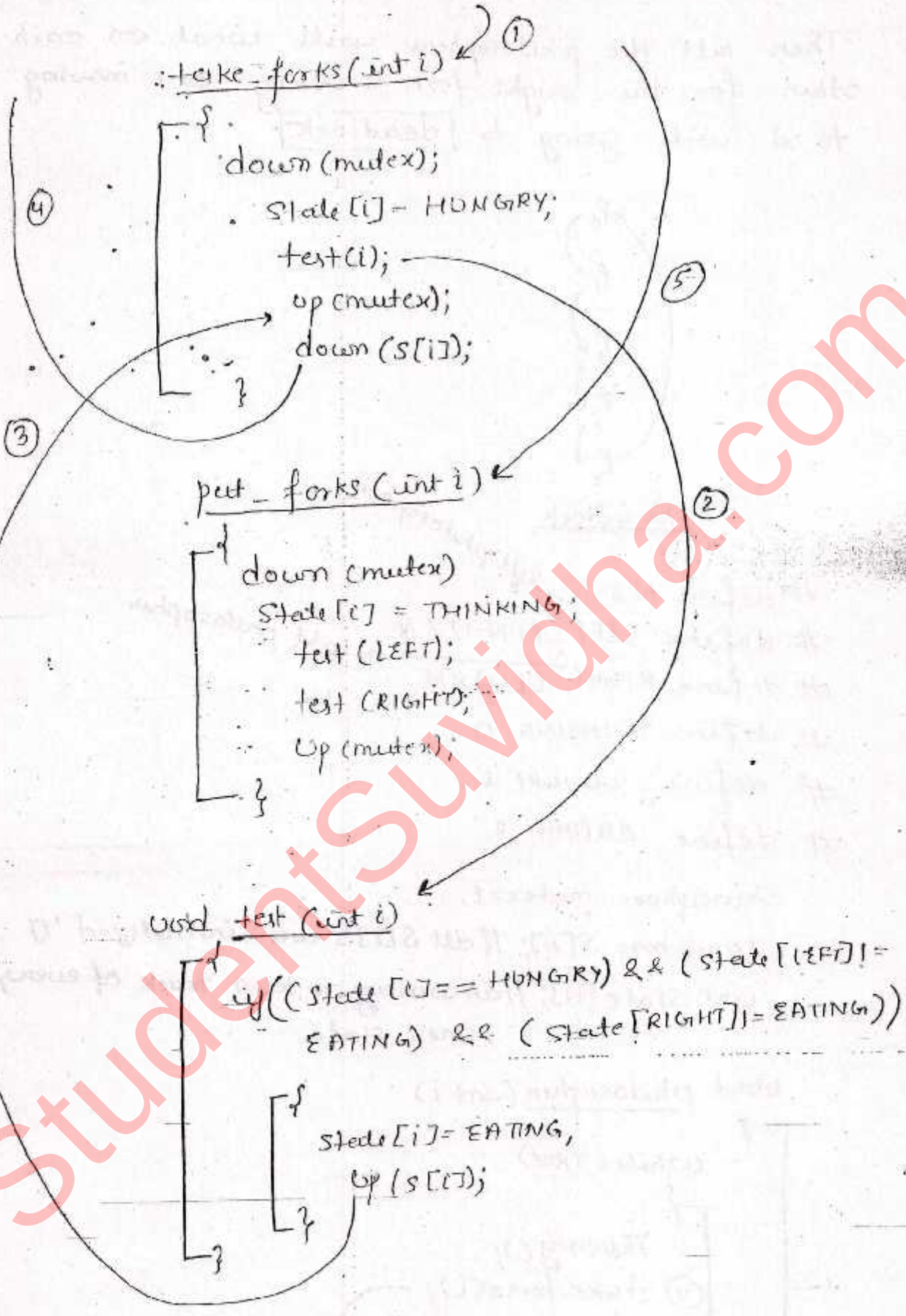
Soln

```
#define N 5                          // left philosopher
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N                // right philosopher
#define THINKING 0
#define HUNGRY 1
#define EATING 2

Semaphore mutex=1;
semaphore S[N]; // all S[i]'s are initialized '0'
int state[N]; // an array to keep track of every one's state

void philosopher(int i)
{
    while(TRUE)
    {
        Thinking();
        take_forks(i);      (4)
        eat();
        put_forks(i);       (5)
    }
}
```

(1)



⇒ mutex is the binary semaphore used by the philosopher in a mutual exclusive manner.

⇒ S[N] is an array of binary semaphore initially all are assign to zero.

⇒ state[N] is an integer array, used to keep track of every philosopher's state.

⇒ initialy all the philosopher will be in the thinking state.

Analysis :-

mutex = 1, 0, 1

P_0 = T S[0] = 0 i = 2

P_1 = T S[1] = 0

P_2 = T, H, E S[2] = 0, 1, 0 .....

P_3 = T S[3] = 0

P_4 = T S[4] = 0

Que. Assume the philosopher P_1 & P_3 are in the eating state, then if the philosopher P_2 is toying to go into eating state, then which statement in the above code is controlling the trouble above situation?

(a) if condition — Check for cond^{n}.....

(b) test(i) condition

(c) down (mutex)

(d) down (S[i]);

Que What happens if you interchange [up(mutex)], [down(S[i])] in the take-forks func.?

(a) no prb, sol^{n} works fine

(b) more than two philosopher will go into eating state, which is wrong

(c) It is possible for deadlock

(d) None of the above.

Sol:-

$P_1 = E$

$P_2 = H$

$P_3 = E$

mutex = 1, 0, -1

S[2] = 0, -1

take_fork(int i)

down[S[i]]

(P_2) get suspended

not perform up(mutex).

put_fork(int i)

down(mutex)

(P_1, P_3) get suspended

Deadlock

if P_0 & P_4 try to eat both are getting suspended by opⁿ down(mutex) bcz mutex = 0

Que Let P[0] ... P[4] be process and m[0] .. m[4] be binary semaphore, mutexes initialized to 1.

Each procer Pi, Executes the below code.

```
wait(m[i]);
wait(m[(i+1) mod 5]);
[c.s.]
signal(m[i]);
signal(m[(i+1) mod 4]);
```

Consider the below statement

I. M.E is satisfied

II. M.E is not satisfied

III. It is possible for deadlock.

Which of the above statements are TRUE?

(a) only I

(b) only II

(c) only I & III

(d) only II & III

Sol:

$P_0 \rightarrow [m_0\ m_1]$ $m_0 = 1$

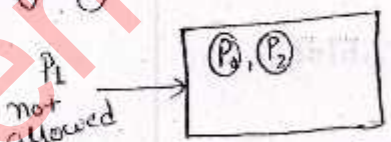
$P_1 \rightarrow [m_1\ m_2]$ $m_1 = 1$

$P_2 \rightarrow [m_2\ m_3]$ $m_2 = 1$

$P_3 \rightarrow [m_3\ m_0]$ $m_3 = 1$

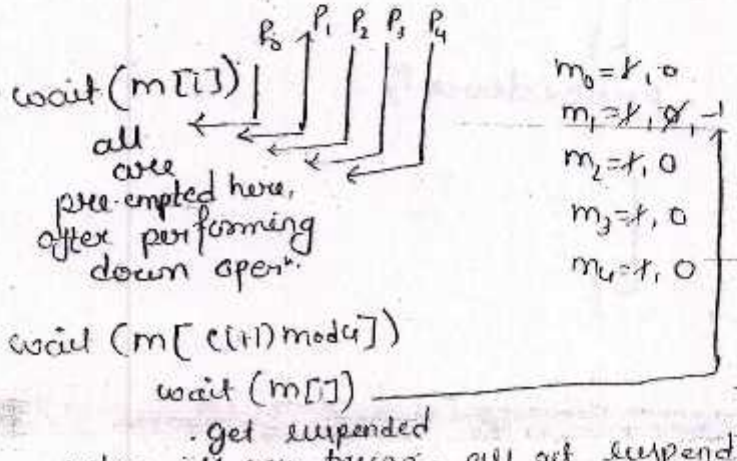
$P_4 \rightarrow [m_4\ m_1]$ $m_4 = 1$

P_0 & P_2 are totally depend on different mutexes. when both trying to enter into CS, both are allowed.



M.E is not satisfied

Deadlock



Deadlock possible

Monitors:-

⇒ Monitors is the programming language, compiler support to achieve synchronization.

⇒ The Monitors is a collection of variables, condition variables and procedures combined together in a special kind of module or a package.

⇒ The processes running outside the monitor can't directly access the internal variable of monitor but however they can called the procedures of the monitor.

⇒ The monitors are an important property that only one process can be inside the monitor at any point of time.

Syntax of the Monitor:

keyword (predefine) — Monitor Example name of the monitor

```
{
    variables;
    condition variables;
    procedure P1
    {
    =
    }
    procedure P2
    {
    =
    }
}
```

⇒ Condition Variables:

The two different operations are performed on the condition variables of the monitor –

1) wait()
2) signal().

```
condition x, y;
```

'x' & 'y' are two condition variable.

wait():

`x.wait()` or `wait(x)`

`y.wait()` or `wait(y)`

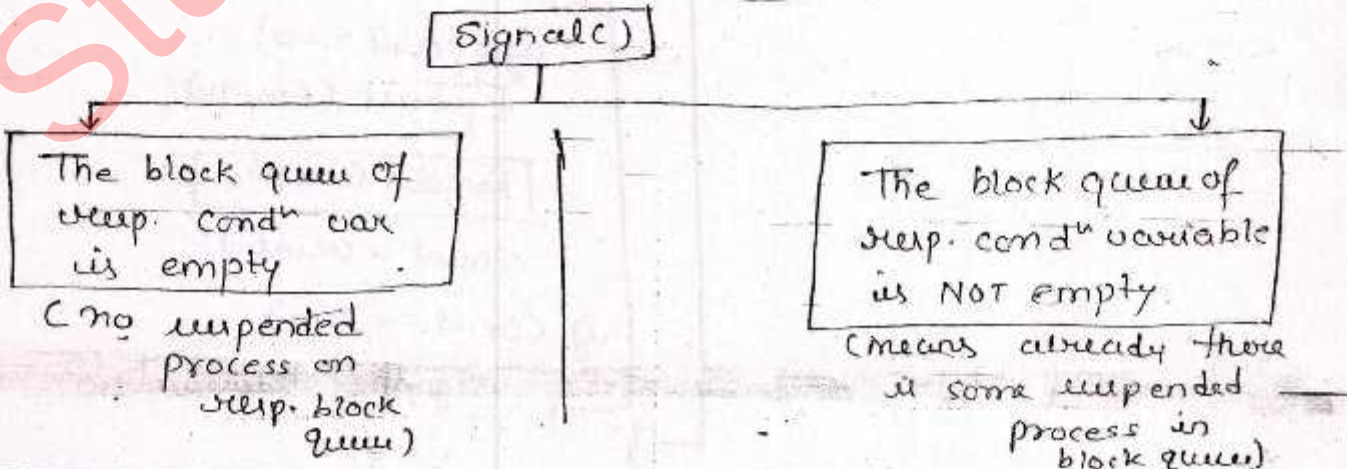
→ The process performing the wait() oper^n on any cond^n variable will be suspended and the suspended process will be placed in the block queue of resp cond^n variable.

→ Every cond^n variable have its own queue. (block queue)

Signal():

`x.signal()` or `signal(x)`

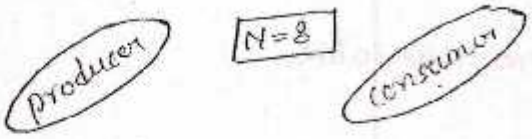
`y.signal()` or `signal(y);` .....

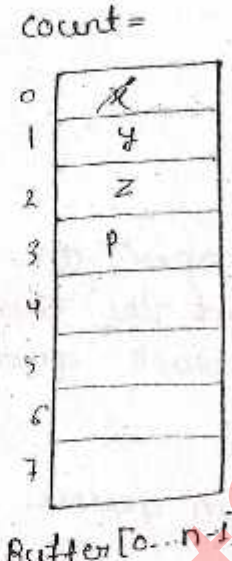


The signal() has no effect, and the signal will be lost.

If process will be resumed from the block queue & any one process will continue the execution.

Producer - Consumer with the monitors :






```
monitor producer_consumer (name of monitor)
{
    integer count = 0;
    condition FULL, Empty;
    producer Enter_items procedure
    {
        if (count == N)
            wait (Full);
        Enter (item);
        Count = count ++;
        if (count == 1)
            Signal (Empty);
    }
    producer Remove_items procedure
    {
        if (count == 0)
            wait (Empty);
        Remove (item);
        count = count - 1;
        if (count == N-1)
            Signal (Full);
    }
}
```

```
procedure producer
{
    while (TRUE)
    {
        produce_item (Itemp);
        producer_consumer. Enter_item ();
    }
}
```

```
procedure consumer
{
    while (TRUE)
    {
        producer_consumer. Remove_item ();
        process_item (Itemc);
    }
}
```

⇒ count is a variable represents the no. of items present in the buffer at any point of time.

Analysis :-

Producer		Consumer
count = 3	Normal execution w/o preemption.	count = 4
Enter item Itemp		Remove item x
count = 4		count = 3

Case I : when buffer is full

Producer
count = 8
producer is suspended & placed in the block queue of full.

Producer

Consumer
count = 8
Remove item.
count = 7
if (count == N-1) (TRUE)
signal (Full)

Producer is resumed from Bqueue.

Case II : when buffer is empty.

Consumer

count = 0
wait(Empty)
Consumer get suspended & placed in Queue of Empty

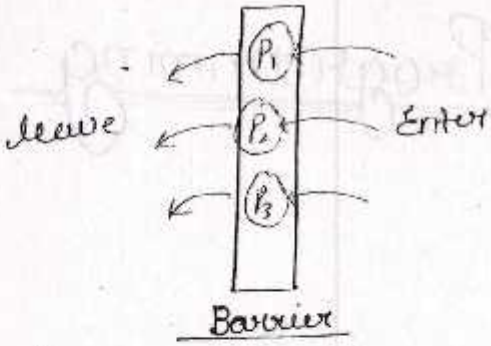
producer

count = 0 —
Enter item
count = 1
signal (Empty)

Consumer will be resumed from block queue.

⇒ Even though, we are using the count variable, there will not be any inconsistency while updating the count variable bcz only one process can be active inside the monitor at any point of time.

Page 42
Date 10-11



Three processes - P_1, P_2, P_3

Initially - $S = 1$

process - arrived = 0

process - left = 0