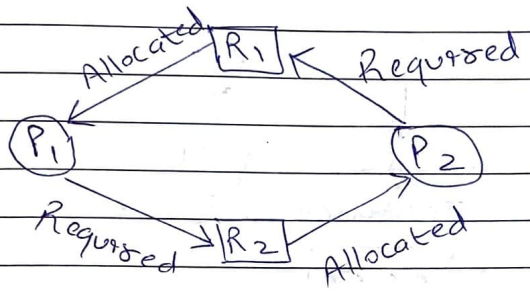
Week-7

Deadlock — (gate smasher) (from gate smasher)

If two or more processes are waiting on happening of some event, which never happens, then we say these processes are involved in deadlock then that state is called deadlock.

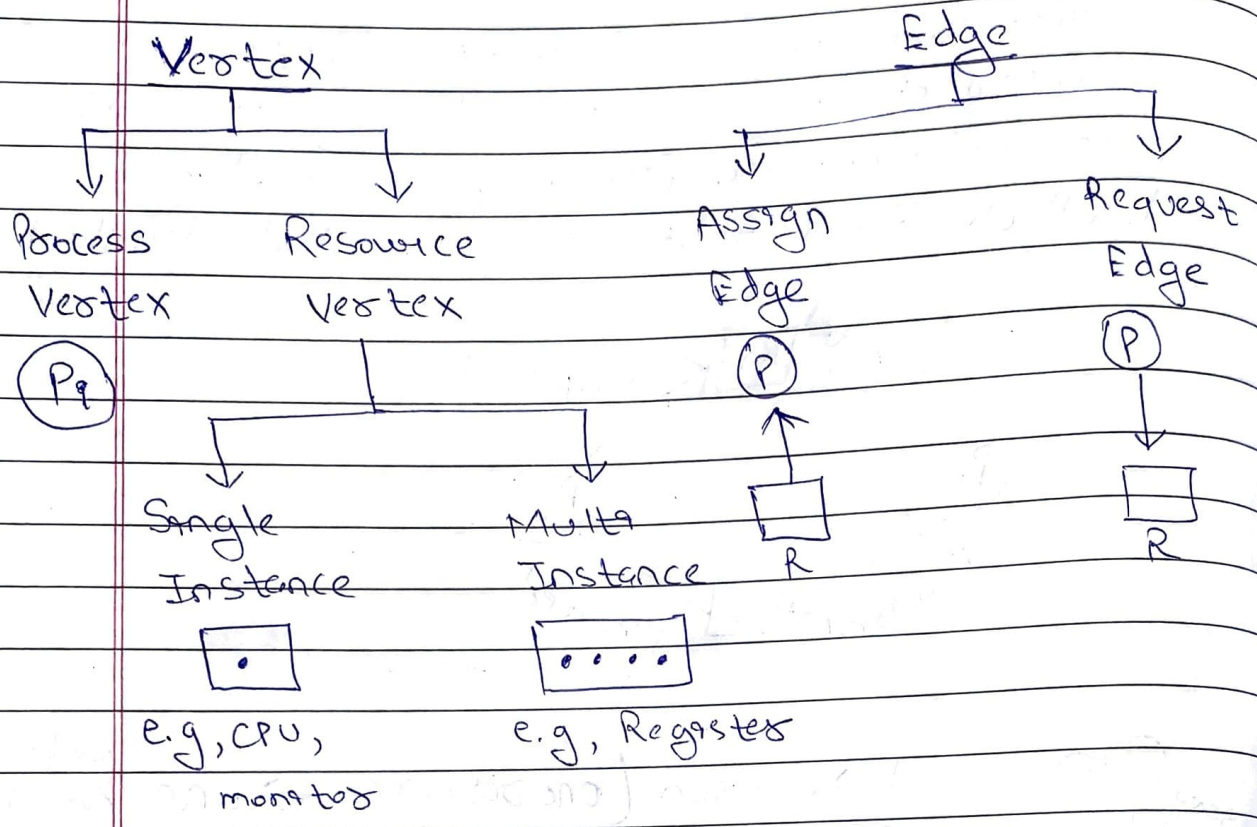


Necessary conditions for deadlock

1. mutual Exclusion (one resource → one only process)
2. No preemption
3. Hold & wait
4. circular wait

(gate smashers)

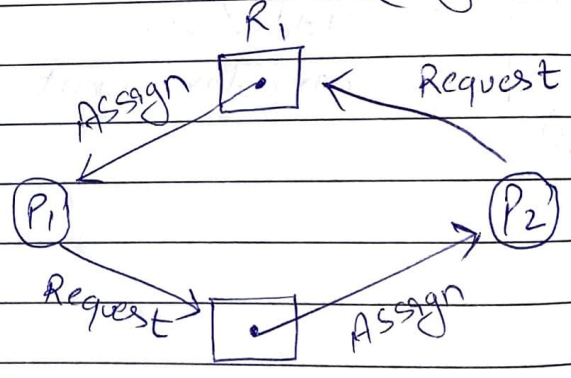
Resource Allocation graph (RAG)



Example 1 –

(single instance)

	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0



Deadlock

Availability (R_1, R_2) = (0, 0)

↳ as it is single instance

Example 2 –

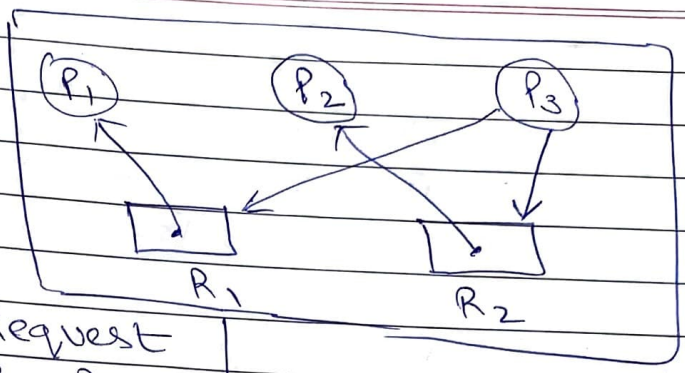
So it will not be able to full fill the request of P_1 and P_2 as P_1 request 0, 1 but availability is 0, 0

Similarly for P_2.

So there would be a deadlock.

Example 2 —

No Deadlock →



	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1

R_1, R_2

Availability (0,0)

for P_1 it is available so now availability will become (1,0)
now also for P_2 it is available as it require (0,0). So now availability will become (1,0) + (0,1) = (1,1)
So now it is also available for P_3
So There will be no deadlock.
(P_1 P_2 P_3) →

(*) Waiting

Finite	Infinite
(Starvation)	(Deadlock)

NOTE

for single instance resources:

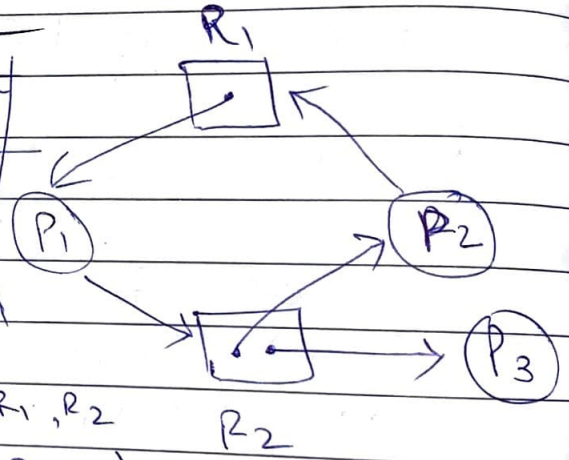
- If RAG has cyclic wait (cycle)
  => Always Deadlock
  → True

- If RAG has no cycle
  => No Deadlock
  → True

Multi-instance RAG (Gate Smasher)

EX-1

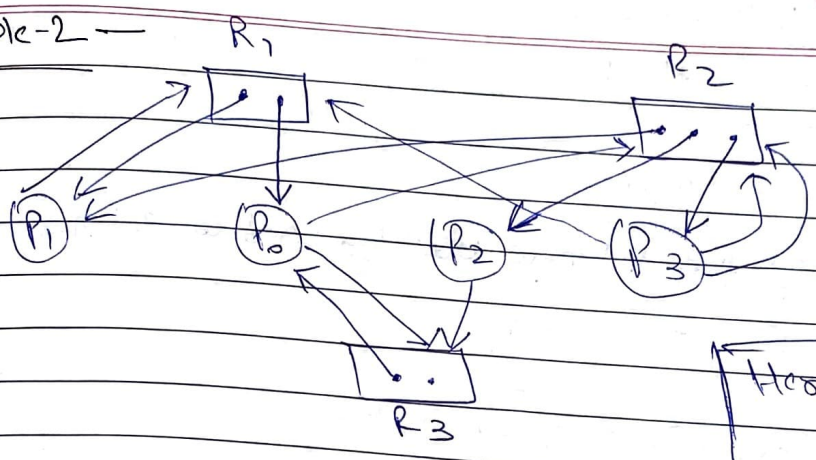
	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0



Current available (0,0)
it can fulfill P_3 ~~Avail~~ Request
P_3 → now avail → 0,0 + 0,1 → (0,1)
it can fulfill P_1 ~~Avail~~ Request
→ now avail → 0,1 + 1,0 → (1,1)
it can fulfill P_2 Request
→ now avail → (1,1) + 0,1
So here is No deadlock.
P_3 P_1 P_2

Multi-instance RAG CW
⇓
Deadlock ⇒ circular wait.
either

Example-2 —



Here, is No Deadlock

		Allocate			Request		
		R_1	R_2	R_3	R_1	R_2	R_3
2X	P_0	1	0	1	0	1	1
3X	P_1	1	1	0	1	0	0
1X	P_2	0	1	0	0	0	1
4X	P_3	0	1	0	1	2	0

Curr. availability → (R_1 R_2 R_3) = (0 0 1)

I can fullfill the request of P_2,
So P_2 will get terminated & whatever resources it is holding will get released
So now curr avail → (0,0,1) + (0,1,0)
→ (0,1,1)

Now → P_0 can full fill
So curr.avail → (0,1,1) + (1,0,1)
→ (1,1,2)

Now, P_1 can full fill
So curr avail. → (1,1,2) + (1,1,0)
→ (2,2,2)

Now, P_3 can fullfill
So, curr avail → (2,2,2) + (0,1,0)
→ (2,3,2)

Sequence → $P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

(gate smasher)

Various Method to handle deadlock

1) Deadlock ignorance (ostrich Method)

~~used widely, this technique uses by windows & Linux.~~

2) Deadlock prevention

- Mutual exclusion → [Every resource will be used by one process at a time]
- No preemption
- Hold & wait
- Circular wait

→ If we ~~try~~ try to false this situation then deadlock will prevent.

3) Deadlock avoidance (Banker's Algo)

4) Deadlock detection & Recovery

Deadlock Question

Q. A System is having 3 processes each require 2 units of resources 'R'. The minimum no. of units 'R' such that no deadlock will occur

(a) 3 (b) 5 (c) 6 (d) 4

⇒

P1	P2	P3
1	1	1
1		

R = 4

this will execute give its resource to P2 then P2 executes & give resource to P3 and so on.

max Req. − 1 → (~~half of~~ ek kam dedo)

max. no. of R → 3

Ex − P1 P2 P3
3 4 5 → require
2 3 4 → 9+1 = 10

Safe Sequence
unsafe

(gatesmasher)

Banker's Algo ——— (Deadlock avoidance / Deadlock Detection)

(Total A=10, B=5, C=7)

Process	Allocation			Max Need			Available			Remaining need			
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	0	1	0	7	5	3	3	3	2	7	4	3	P_1
P_2	2	0	0	3	2	2	5	3	2	1	2	2	P_2 ✓
P_3	3	0	2	9	0	2	7	4	3	6	0	0	P_3
P_4	2	1	1	4	2	2	7	4	5	2	1	1	P_4 ✓
P_5	0	0	2	5	3	3	7	5	5	5	3	1	P_5
	7	2	5				10	5	7				

A → CPU, B → Memory
C → Printer

Current = 10−7, 5−2, 7−5
3, 3, 2

Remaining need = Max − Allocation

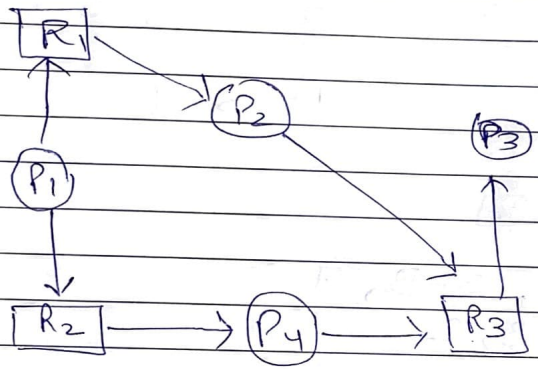
$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

Safe sequence (as Deadlock does not occur)

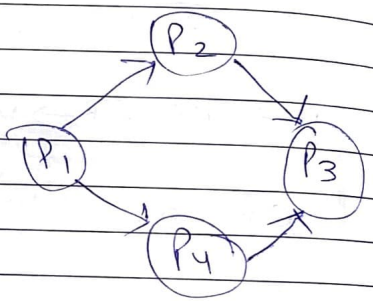
* If we cannot full fill any process need then we can say there ~~must~~ will be deadlock.
* We will go from top to bottom then again start from top.

Deadlock Detection & recovery

- Deadlock detection Algorithm (2 types)
  - Single instance (wait for graph)
  - Multiple instances (Bankers algo)

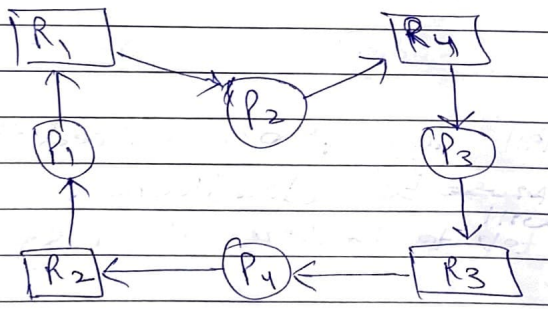


(RAG)

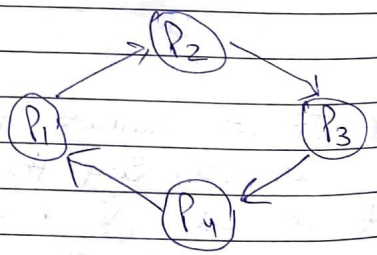


(wait for graph)

(no Deadlock)



(RAG)
(Resource allocation graph)



(wait for graph)

(Deadlock)

Bankers's

	Allocation		
	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Total

P_0

Q. Consider o
Share 4
process ca
The large
deadlock

Reso

(*) Banker's algo (Safety algo)

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2	0	1	0
P_2	3	0	3	0	0	0	3	1	3
P_3	2	1	1	1	0	0	5	2	4
P_4	0	0	2	0	0	2	7	2	4
							7	2	6

Total A B C
7 2 6

P_0 P_2 P_3 P_1 P_4

Q. Consider a system with 3 processes that share 4 instances of same resource type. Each process can request a max. of 'K' instances. The largest value of K that will always avoid deadlock is ___.

⇒ (*) Resource + Processes > total demand

→ no. of processes

4+3 > 3×2 → Providing 2 instances to each processes.

7 > 6 True

4+3 > 3×3

7 > 9 False

Module – 3

Process synchronization —

A Cooperating ~~system~~ process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

- directly shares a logical address space (i.e, code and data)
- or be allowed to share data only through files or messages

Concurrent access to shared data may result in data inconsistency.

Producer Consumer problem —

A producer process produces information that is consumed by a consumer process.

- One solution to this problem uses shared memory
- To allow pro. and cons. proce. run concurrently we must have available a buffer of items that can be filled by producer and emptied by consumer.

Two kind of Buffer

- Unbounded buffer
  - no practical limit on size of buffer
- Bounded buffer
  - Assumes buffer size is fixed.

EX- • Counter is currently 5
• Pro. & con. pro: executes the statements "counter++" and "counter--" concurrently
• value of variable counter may be 4, 5 or 6
• only current result is counter == 5 generates when pro. & con. executes seperately.

Counter++ may be implemented in machine language as:

register$_1$ = counter
register$_1$ = register$_1$ + 1
counter = register$_1$

Counter-- as:

register$_1$ = counter
register$_1$ = register$_1$ − 1
counter = register$_1$

T_0	Pro. executes	register$_1$ = counter	{register$_1$ = 5}
T_1	Pro.	register$_1$ = register$_1$ + 1	{register$_1$ = 6}
T_2	con.	register$_2$ = counter	{register$_2$ = 5}
T_3	con.	register$_2$ = register$_2$ − 1	{register$_2$ = 4}
T_4	Pro.	counter = register$_1$	{counter = 6}
T_5	con.	counter = register$_2$	{counter = 4}

→ This is race condition.
Clearly, we want the resulting changes not to interfere with one another, Hence we need Process Synchronization.

(Education 4u) (utube channel)

Process Synchronization

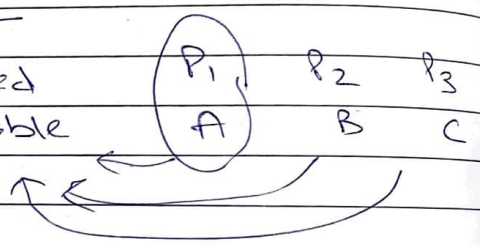
2 types

Independent process
"Execution of one process does not effect the execution of others"

Cooperative process
"effect the execution of others"

Critical section problem —

Code Segment ⇒ Shared Variable




```
do
{
    Entry Section
        Critical Section
    Exit Section
        Remainder Section
} while (true);
```

Entry section → controls th entry into Critical Section & get lock on required resources

Exit section → removes lock on resources & others know that its Critical Section is over

Solution

Must s

(1) Mutual
If a p
other p

(2) Progres
These
that h

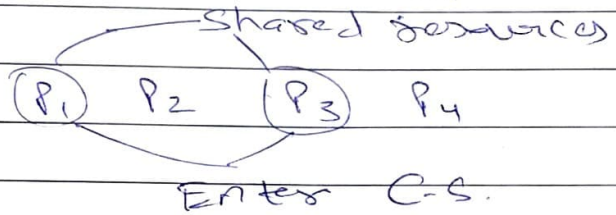
(3) Bounded

Solution to C.S. Problem —

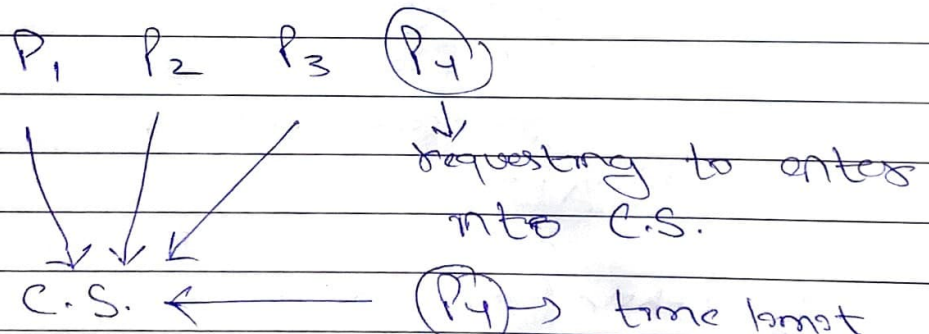
Must satisfy three condition :

(1) Mutual exclusion :
If a process is executing in its C.S, then no other process is allowed to execute in C.S.

(2) Progress :
Those process that want to enter into C.S. that has to be allowed.



(3) Bounded waiting :



Peterson's Solution —

- A classical software-based solution to the critical-section problem.
- Two processes share two variable —

int turn;	bool flag[2];
indicates whose turn it is to enter its critical section	used to indicate if a process is ready to enter its critical section

Algorithm for process P_0 —

```
do {
    flag[i] = true;
    turn = j;
    while ( flag[j] && turn == j );
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

Test and

- A Hardware
- There is sh
- If lock →
- If not lo

```
boolean
    boole
    * tar
    retu
}

do {
    while (
    // crit
    lock =
    // rem
} whil
```

Swap ins

```
void S
{
    bo
}
```

Solution
Shared

Set

Test and lock —

- A Hardware Solution to the synchronization
- There is shared lock variable i.e, 0 or 1
- If lock → waits
- If not locked → executes the critical section

```
boolean TestandSet( boolean * target) {
    boolean rv = * target;
    * target = True;
    return rv;
}

do {
    while ( TestandSet(&lock)) ; → busy waiting
    // critical section
    lock = false
    // remainder Section
} while (true);
```

Swap instruction —

```
Void Swap ( boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Solution using Swap() :—

- Shared boolean variable lock initialized to false.

- Each process has a local boolean variable key
- Solution For mutual Exclusion:

```
do {
    key = true;
    while ( key == true) → busy waiting
    Swap (&lock, &key);
     critical section
    lock = false;
    Remainder section
} while(true);
```

C.S. Sol. using "Test_and_set" instruction — (Gate smasher)

```
while (test_and_set(&lock));
CS
lock = false;
```

```
boolean test_and_set(boolean
                     *target)
{
boolean r = *target;
*target = True
  return r;
}
```

lock: false → 1000 → True

target: 1000 → false → True

r: false

True

True → False → True

1. while (lock == 1);
2. Lock = 1
3. Critical section
4. Lock = 0

P_1	P_2
1 * 2	1
↓	
preempt hogaya	
(P_1 emergency aagya)	

P_2 ← P_1 ← done aaGaya C.S. m. two mutual exclusion

L = ∅ ∧ 1

Semaphore — (gate smasher)

Counting (−∞ to +∞)

Binary (0, 1)

"Semaphore is an integer variable which is used in mutual exclusion manner by various concurrent cooperative processes in order to achieve synchronization".

P(), Down, wait → (multiple synonyms)

V(), up, signal, post, release

Entry code: CS

exit code

(P, or wait)

```
Down ( Semaphore S )
{
    S.value = S.value − 1
    if ( value < 0)
    {
    Put Process (PCB) in
    Suspended list,
    Sleep()
    }
    else
    return;
}
```

P1 P2 P3

Entry Section

CS P1, P2, P3

S

~~S=0~~ ~~X X ≠ ≠~~ −2

(intially taking S=3)

Block list

P4 P5

S=0 → no process in block list

S=−4 → 4 process in block list

(V, signal, post)

```
Up( Semaphore S)
{
  S.Value = S.value + 1;
  if( S.value ≤ 0)
  {
    Select a process
    from suspended list
    & wakeup()
  }
}
```

S

3	2	1	0	-1	-2

Block list

P4	P5

Now these process will release from Block list and are ready to enter in entry section.

(*) S = 10

how many successfull operation can be perform

Ans ⇒ (16)

(C.S. ~~Kar~~ ~~andar~~ aayega)

(*) S = 10, 6P, 4V

what will be the final value of Semaphore?

⇒ 10 − 6 ⇒ 4

4 + 4 ⇒ 8 → ~~final value~~

(*) S = 17, 5P, 3V, 1P

⇒ 17 − 5 + 3 − 1

⇒ 14

(Gate smashers)

(Same as mutex lock)

Binary Semaphore

Down(Semaphore S)	up (Semaphore S)
{	{
if (S.value == 1)	if (suspendlist is empty)
{	{
S.value = 0;	S.value = 1
}	}
else {	else {
Block this process and place in suspend list, sleep();	select a process from suspend list and wake up();
}	}
}	}

Down, P, wait

up, V, signal

S = 1 0

Successfull operation

S = 0 0 Block

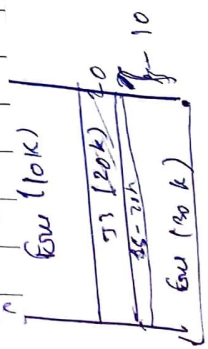
unsuccessfull operation

Ex

S = 1 0

P_1	P_2
Down (S)	Down (S)
CS	CS
up (S)	up (S)

Let P_2 start first,
P_2 will enter C.S.
P_1 will get block.
P_2 will get an up
up will set P_1 in ready queue



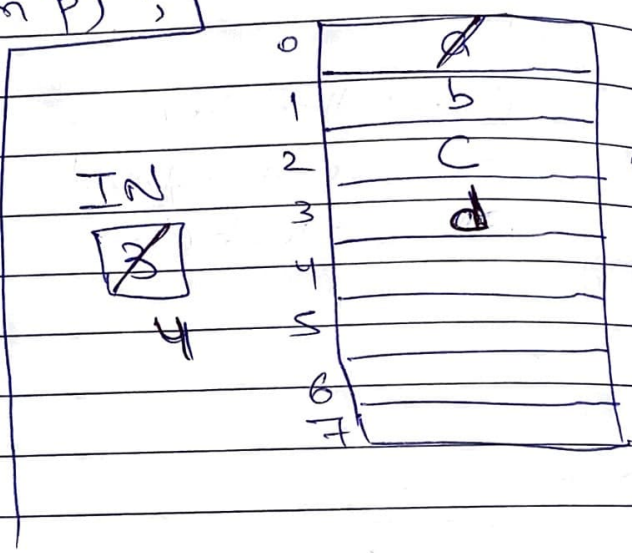
(Bounded-buffer Problem)

Solution of Producer Consumer Problem using Semaphore :-

Counting Semaphore → full = 0 = No. of filled slots
→ empty = N = No. of empty slots.

Binary semaphore S = 1.

```
Produce_item (item p);
1) down (empty);
2) down (S);
   Buffer [IN] = Item p;
   In = (In+1) mod n;
3) up (S);
4) up (full);
```




```
Consumer
1) down (full);
2) down (S);
   item c = Buffer [out]
   out = (out+1) mod n;
3) up (S)
4) up (empty);
```

item c = Buffer[0] = a

out will increment

Case I → No Prompt

empty = 5 ✗ 5
full = 3 ✗ 3

So to check consistency we do 5+3 = 8

S = ✗ 0 ✗ 0 1

CASE II — (Prempt)

first Producer run only 1st command then prempt and consumer run all its command then again producer will run from 2nd command.

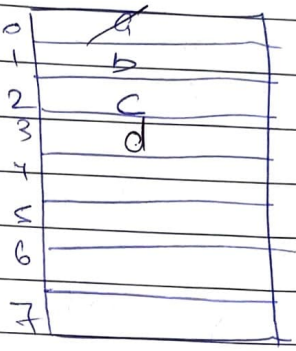
empty = ~~0~~ 4 5

Full = ~~3~~ 2 3

S = ~~1~~ 0 ~~1~~ 0 1

IN = 5 4

OUT = 0 1



5+3 = 8 = no of slots

(empty + Full) =

(gate smashers)

Solution of Readers-Writers Problem using Binary Semaphore

```
int rc = 0
Semaphore mutex = 1;
Semaphore db = 1;
Void Reader (void)
{
  while (true)
  {
    down(mutex);
    rc = rc + 1
    if(rc == 1) then down(db);
    up(Mutex)
    DB
```

Entry code

```
    down(Mutex)
    rc = rc - 1
    if(rc == 0) then up(db);
    up(Mutex)
    Process_data;
  }
}
Void writer (void)
{
  while(true)
  {
    down(db);
    DB
    up(db);
  }
}
```

Exit code

R-W → Problem
W-R → Problem
W-W → Problem
R-R → Problem

rc → read count
no. of readers in buffer.

CASE I
rc = 0
mutex = 1
db = 1
DB R1

CASE II
rc = 0 1
mutex = 1
db = 1

CASE III
rc = 0
Mutex = 1
db = 1

CASE IV
rc = 0
mutex = 1
db = 1
If pro

CASE I — (R–W)

(R1 first come then W1 comes)

rc = ~~0~~ 1

mutex = ~~1~~ 0 ~~1~~

db = 1 0

DB R1

down(db)

Not possible as db = 0.

∴ writer will block.

CASE II — (W–R)

rc = 0 1

mutex = 1 0

db = 1 0

first

DB W1

now Reader comes

down(db) not possible

because db is already zero.

∴ Reader will block.

CASE III — (W–W)

rc = 0

Mutex = 1

db = 1 0

DB W1

second writer will block

as db = 0.

CASE IV — (R–R)

rc = 0 1 2 3 2 1

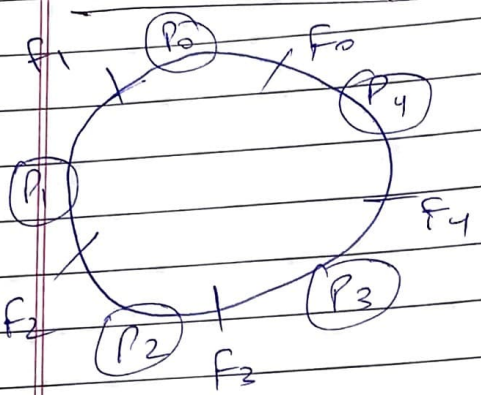
mutex = 1 0 1 0 1 0 1 0 1 0 1

db = 1 0

DB R1 R2 R3

If process done & R1 want to exit,

Dining Philosopher's problem

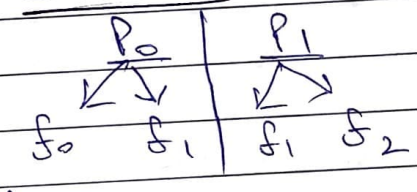


5 philosophers
5 forks

```
Void Philosopher (void)
{
  while(True)
  { Thinking();
    take_fork(i);  → left fork
    take_fork((i+1)%N);  → Right fork
    EAT();
    Put_fork(i);
    Put_fork((i+1)%N);
  }
}
```

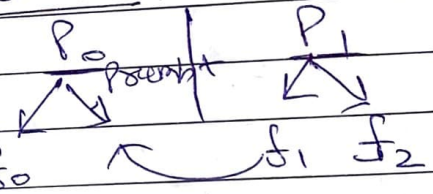
(har Philosopher sbse phele left wala fork uthata hai fir right wala)

Case I —



f(o+1) mods
1 mods
①

Case II —



To solve this we use Semaphore,

Solution using Semaphore —

S[5] for 5 Semaphores

```
Void Philosopher(void)
{
    while(True)
    {
        Thinking();
        wait(take_fork(Si));                 } Entry code
        wait(take_fork(S(i+1) mod N));       }
        EAT();
        Signal(Put_fork(Si));                } exit code
        Signal(Put_fork(S(i+1) mod N));      }
    }
}
```

S_0	S_1	S_2	S_3	S_4
↓	↓	↓	↓	↓
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
0	0	0	0	0

✓ P_0	:	S_0	S_1
✓ P_1	:	S_1	S_2
✓ P_2	:	S_2	S_3
✓ P_3	:	S_3	S_4
✓ P_4	:	S_4	S_0

Deadlock

→ P_0 aaya left wala fork uthaya aur preempt hogaye

Phir P_1 aaya ——————————

Phir P_2 aaya —————— " ——————

Phir P_3 aaya —————— " ——————

Phir P_4 aaya —————— " ——————

now,

→ P_4 kehe raha hai S_0 do, par $S_0 \to 0$ hai toh P_4 block ho jayege.

Similarly P_3 wants S_4, but P_4 $S_4 \to 0$ so P_3 block

Similarly all process will block.

→ Now this condition is deadlock.

→ **Solution to remove deadlock:**

✓ P_0	S_0	S_1
✓ P_1	S_1	S_2
✓ P_2	S_2	S_3
✓ P_3	S_3	S_4
block P_4 →	S_0	S_4

S_0	S_1	S_2	S_3	S_4
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
0	~~0~~	~~0~~	~~0~~	0
	1	1	1	

→ swap the last process right fork phele uthayega aur left wala baad mai