10.02.2025

* Process synchronization

① Data synchronization
② control synchronization

shared = 4 (common variable)

	Process P_0		Process P_1	
case 1 -	A1 : reg 1 + shared ;	4	B1 : reg 2 = shared ;	4 7
case 2 -	A2 : reg 1 = reg 1 + 3 ;	4+3	B2 : reg 2 = reg 2 - 3 ;	4-3 7-3
	A3 : shared = reg 1	7	B3 : shared = reg 2 ;	1 4
	shared = 7		shared = 1	

↳ ① Independently (case1)
P_0 → shared = 7
P_1 → shared = 1

② sequencially (case 2)
P_0 → access first P_1 → access first (case 3)
shared = 7 shared = 1
P_1 → shared = 4 P_0 → shared = 4

→ when a same code accessed again & again, generating different output → race condition.
type of process (share same variable & resource)
↳ concurrent process.
Issues in Data synchronization

→ The two process are known as 'concurrent process' if they share same resource at the same time.
→ Process synchronization has 2-types
① Data synchronization
② control synchronization.

→ case 1 : Independently
when 2 process are executed independently:
Initially shared = 4
when P_0 gets executed, it makes shared = 7.
when P_1 gets exeacted, it makes shared = 1

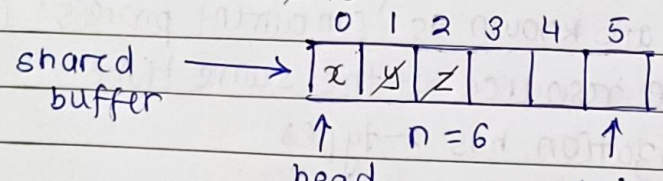
case 2:
when the process are executed sequencially & P_0 is executed first, it makes shared = 7
& when P_1 (executed after P_0) it makes shared = 4.

case 3:
when the process are executed sequencially & P_1 is executed first; it make shared = 1 &
when P_0 (executed after P_1) it makes shared = 4.

→ Looking into case 1, case 2 & case 3, the processes P_0 & P_1 are executing the same code multiple times but generating different results.
→ when the process executes the same code again & again but generates different result at every input such condition is known as 'Race condition'.

② control synchronization

P_0 → Producer	P_1 → consumer
fills all empty cells	Empty all filled cells.
E = 6	F = 0 initially



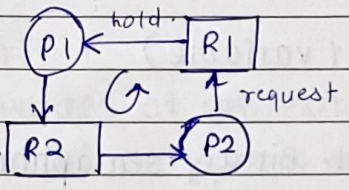
control → P_0 / P_1

* Critical section

```
do {
   while (1)
   {
      entry section ();
      critical section ();
      exit section ();
      remaining section ();
   }
}
```

→ conditions for critical section :-

1. Mutual exclusion – single time → one process
2. Progress – If a process wishes to go into a critical section, then only ...
3. No deadlock –
   * cycle is been created;



process holding resource
∴ arrow from resource to process

entry & ending process is same.

$P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1$

4. Bounded wait – process cannot remain in critical section for a long time.

→ - Critical section is a small piece of code which is executed by many processes at several times.
- It is used to avoid the 'race condition'.
- The process should satisfy the following conditions before entering into a critical section.

1- mutual exclusion
- In this only one process can execute into a critical section at a single time.

2- Progress
- If a process wishes to enter into a critical section then only the system allows the process to enter into a critical section.

3. No deadlock
- A process can enter into a critical section which is not into a deadlock state

4. Bounded wait
- No process can wait into a critical section for a longer period of time.

* Semaphores (variable)

Types — 1. Binary semaphores (0 & 1)
2. counting semaphores (0 to +∞)

Operations — 1- wait → dec value of semaphore by 1 (S--)
2- signal — inc value of semaphore by 1 (S++)

```
do {                    Initialize S=1
  while (1)
  {
    entry section();    S--    S=0
    critical section();
    exit section();     S++    S=1
    remaining();
  }
}
```

→ semaphores are the +ve integer variables which resides at entry & exit section of the critical section.

→ Types of semaphores :-

1. Binary semaphores
   In this, the semaphore value will be 0 or 1

2. Counting semaphores
   In this, semaphore value will be 0 to $+\infty$. (or more than 0 to +ve integer value.)

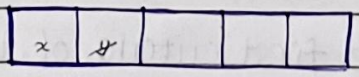
→ Operations of semaphores :-

1. wait operation
   This operation resides at the entry section where it decrements the value of semaphore by 1 (S--)

2. signal operation
   This operation resides at the exit section, where it increments the value of semaphore by 1 (S++).

* Producer-consumer problem

(binary semaphore) S=1



n=5 (empty cells) E=5
(filled cells) F=0

E & F → counting semaphores.

Producer()		Consumer()	
while(1) {		while(1) {	
wait(E);	4 3	wait(F);	
Entry(S) → wait(S);	0	wait(S); ← entry	
CS → append();	stuck	consume(); ← CS	
Exit → signal(S);	1	signal(S); ← exit	
signal(F);	2	signal(E);	
}		}	

17.02.25

* Two solution problem

initially turn = 0 (x ≠ 0)

case 1 :-

P0	P1
while(1)	while(1)
{	{
while (turn != 0);	while (turn != 1);
CS;	CS;
turn = 1;	turn = 0;
RS;	RS;
}	}

✓ mutual exclusion
2) progress
3) bounded wait
4) no deadlock

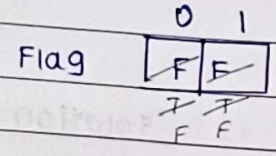
→ Initially, variable turn = 0.

- Process P0, wants to enter into a critical section hence, it checks the statement while (turn != 0); which is False & hence, P0 enters into a CS.
- After the entire execution, P0 makes turn = 1 & exits.

- Process P1 wants to enter into a critical section. It executes the condition while (turn != 1); which is False & hence P1 enters into a CS.
- After the entire execution, P1 makes turn = 0 & exits.

- In this case, the first criteria of the CS ie mutual Exclusion is satisfied but it fails to satisfy Progress condition / criteria.

case 2 :-



P0	P1
while (1)	while (1)
{	{
Flag [0] = T;	Flag [1] = T;
while (Flag [1]);	while (Flag [0]);
CS;	CS;
Flag [0] = F;	Flag [1] = F;
RS;	RS;
}	}

→ Initially, a flag is introduced with both 0 & 1 as false. (same as before).

* Reader - writer problem.

For writer	For reader
wait (wrt);	wait (mutex);
write operation;	readcount ++;
signal (wrt);	if (readcount == 1)
	wait (wrt);
	signal (mutex);
	read operation ();
	wait (mutex);
	readcount --;
	if (readcount == 0)
	signal (wrt);
	signal (mutex);

wrt = 1
mutex = 1
readcount = 0

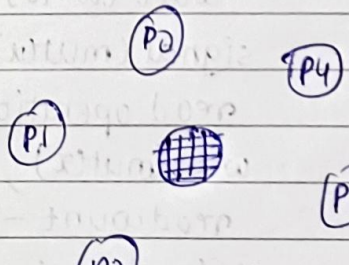
* Dining-philosopher problem

solution 1

```
void philosoper (int i)
{
  while(1)
  {
    think();
    take(Ri);
    take(Li);
    eat();
    put(Li);
    put(Ri);
  }
}
```

solution 2

```
void philosoper (int i)
{
  while(1)
  {
    think();
    take(Ri);
  if (available(Li))
  {
    take(Li);
    eat();
    put(Li);
    put(Ri);
  }
  else
  {
    put(Ri);
    sleep();
  }
}
}
```



solution 3

```
void Philosopher (int i)
{
  while(1)
  {
    think();
    lock(mutex);
    take(Ri);
    take(Li);
    eat();
    put(Li);
    put(Ri);
    unlock(mutex);
  }
}
```

03-03-25

Date:

3- DEADLOCK

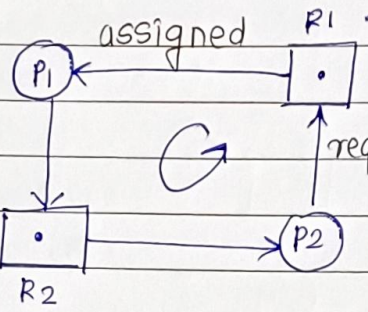
(P) = Process vertex

[R] = Resource vertex

(P) ← [R] = Allocated Edge / Holding / Assigned

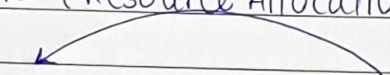
(P) → [R] = Requesting Edge / ~~Hold~~

eg:-



If the cycle is present, deadlock exists.

→ RAG (Resource Allocation graph)



(cycle exists)

→ Deadlock is defined as 'permanent blocking' of a set of processes that either compete for the system resources or communicate ~~with~~ for each other.

- In multi-processing system, several processes request for various resources.

→ RAG (Resource Allocation graph) :-

- It is a graphical representation of the process & the resources to identify or determine the deadlock is present or not & state whether the system is in safe or unsafe state.

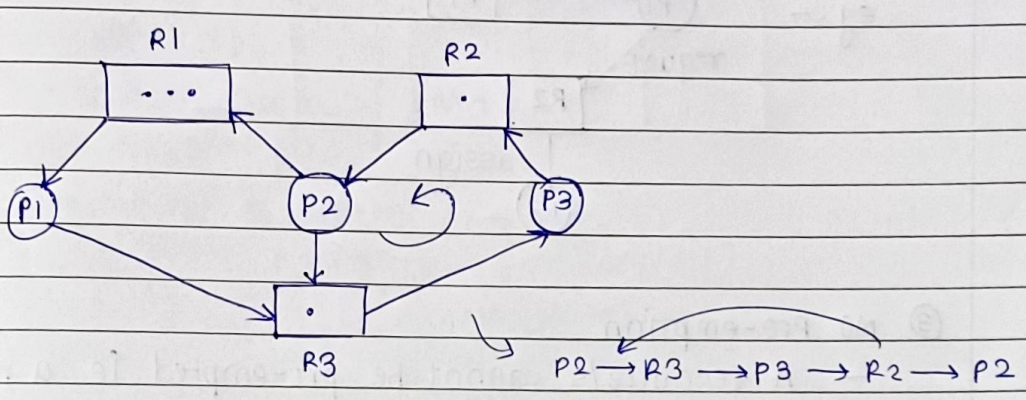
- 2 vertices ie process vertex & resource vertex
- 2 edges

- If the edge is going from resource to process, then it indicates that it is the 'Allocated' edge.

- If the edge is going from process to resource, then it indicates that it is the 'Requesting' edge.

eg:- (from prev diag)

(8)



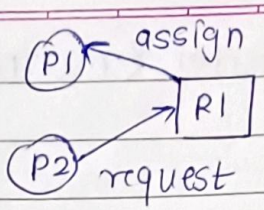
~~P1 → R3 → P3 → R2 → P2 → P1~~

* conditions for deadlock

① Mutual Exclusion
② Hold & wait
③ No pre-emption
④ Circular wait

① Mutual Exclusion
- It works on non-shareable resources
- In this, no 2 processes can execute same resource at the same time.

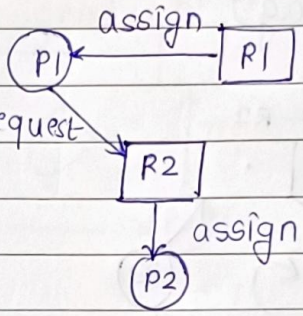
eg :-



② Hold & wait

- There must exist a process that is holding atleast one resource & is waiting to get allocated additional resources that are currently been hold by some other process.

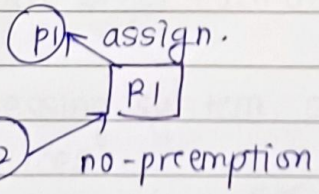
eg :-



③ No Pre-emption

- A resource is cannot be pre-empted ie a resources cannot be released by a process who is holding it.
- It can be released only when the process completes it's execution.

eg :-



④ circular wait

- There must be set of processes ie P_1 to P_m requesting the resources, such that P_1 requests resource R_2 which is held by process P_2 & P_2 requests resource R_1 which is held by process P_1.

- In this case, when a process requests some other resource, which is been held by some another process, then circular wait occurs.

* Banker's Algorithm

1. Deadlock Detection — RAG
2. Deadlock Prevention
3. Deadlock Avoidance — Banker's Algorithm
4. Deadlock Recovery

① Maximum Matrix
② Allocated matrix } given
③ Available matrix
④ Need matrix ← calculated.

$$P = Max(i,j) - Allocated(i,j)$$

(i: process no., j: resource no.)

if Need $(i,j) \leq$ old available (i,j)
then,
New available (i,j) = old available (i,j) + Allocated (i,j)

8)

	R1	R2	R3
P1	4	5	1
P2	2	2	1
P3	1	2	2

maximum matrix

	R1	R2	R3
	7	10	5

Available matrix

System – Initial available

	R1	R2	R3
P1	2	2	0
P2	1	1	0
P3	1	0	0

Allocated matrix.

→ (next page →)

R1 R2 R3

New Available = [7-(2+1+1)] [10-(2+1+0)] [5-(0+0+0)]
= [3] [7] [5]
R1↑ R2↑ R3↑

New Available =

R1	R2	R3
3	7	5

Need matrix = Max(i,j) − Allocated (i,j)

	R1	R2	R3
P1	2	3	1
P2	1	1	1
P3	0	2	2

For P_1 ; Available ~~row~~ matrix row > Need matrix row

Need (P_1) Available (P_1) Allocated (P_1)
(2,3,1) ≤ (3,7,5) (2,2,0)

Add available + allocated for (P_1)
New available = (3 7 5) + (2 2 0)
= (5 9 5)

For P2 ; ~~For~~ Available(~~P3~~) Need (P_2)
(5,9,5) ≥ (1,1,0)

New Available = (5 9 5) + (1 1 0)
= (6 10 5)

∴ system is in safe state
<P_1, P2, P3>

For P3; Available (P_3) Need (P_3)
(6,10,5) ≥ (0,2,2)

New Available = (6, 10, 5) + (1 0 0)
= (7 10 5)

8) Max

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocate

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Initial Available matrix

R1	R2	R3
9	3	6

calculate :-
1. New available
2. Need
3. check each process.

→

New Available = [9-(1+6+2+0)] [3-(0+1+1+0)] [6-(0+2+1+2)]
= [0] [1] [1]

New Available =

R1	R2	R3
0	1	1

Need matrix = max (i,j) – Allocated (i,i)

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

For P1 ; Available (P1) ≤ Need (P1)
(011) ≤ (222)

For P2 ; Available (P2) ≥ Need (P2)
(011) ≥ 001

New available = (0 1 1) + (6 1 2)
= [6 2 3]

P1; Available (P1) ≥ Need
(6, 2 3) ≥ (2 2 2)

New Available = (6 2 3) + (1 0 0)
= [7 2 3]

P3; Available (P3) ≥ Need
(7 2 3) ≥ (1 0 3)

New available = (7 2 3) ~~(1 0 3)~~ (2 1 1)
= ~~[8 2 6]~~ [9 3 4]

P4: Available (P4) ≥ Need
(9, 3, 4) ≥ (4 2 0)

New available = (9 3 4) + (0 0 2)
= [9 3 6]

∴ System is in safe state < P2, P1, P3, P4 >