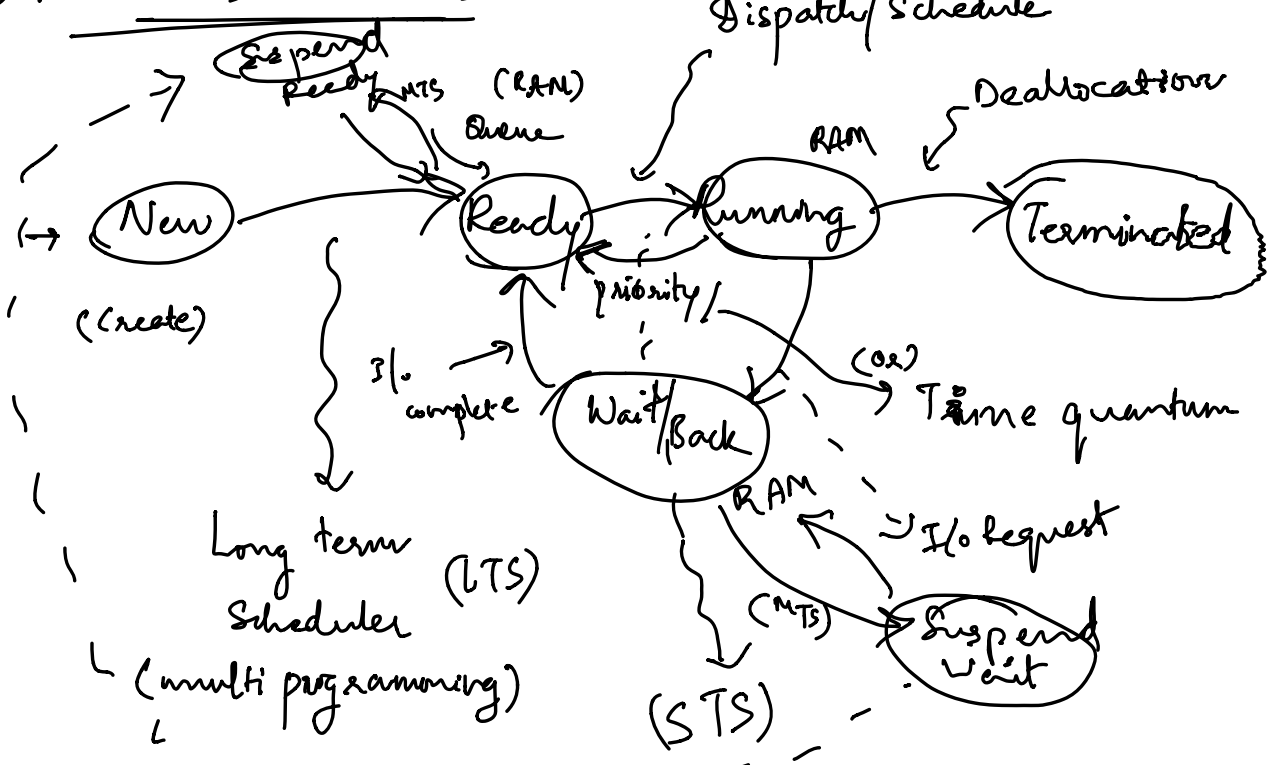
① Process States



Non-preemptive → if CPU runs a process completely without terminating i.e., no priority/time quantum stops else preemptive

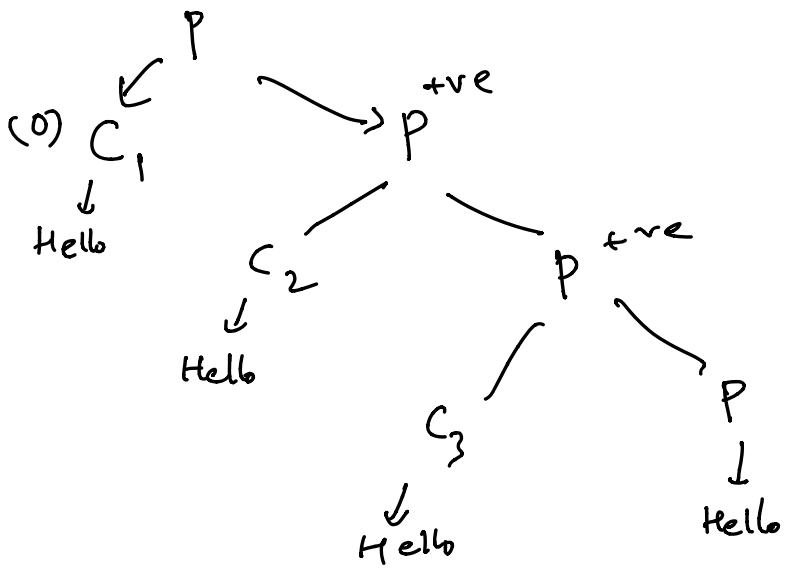
---

② Fork

```
main(){
    if (fork() && fork())
        fork();
}
```

⇒ P (parent process)

```
pf("Hey");
return 0; }
```



---

(3) CPU Scheduling

→ Arrival: when ps enters ready state

→ Burst: time required by a ps to get execute on CPU

→ Completion time: time at which ps completes its execution.

→ turnaround time:
(completion − arrival)

→ waiting time:

(turnaround − burst)

Ps have CPU bound & I/O bound

ps → CPU → I/o req → CPU

→ Response time:

(time at which ps gets CPU first time − arrival time)

---

1) FCFS → criteria: Arrival time
mode: non-preemptive

2) SJF → criteria: Burst-time
mode: non-preemptive

3) SRTF → criteria: Burst-time
mode: pre-emptive

(SJF with pre-emption)

(gantt chart = running queue)

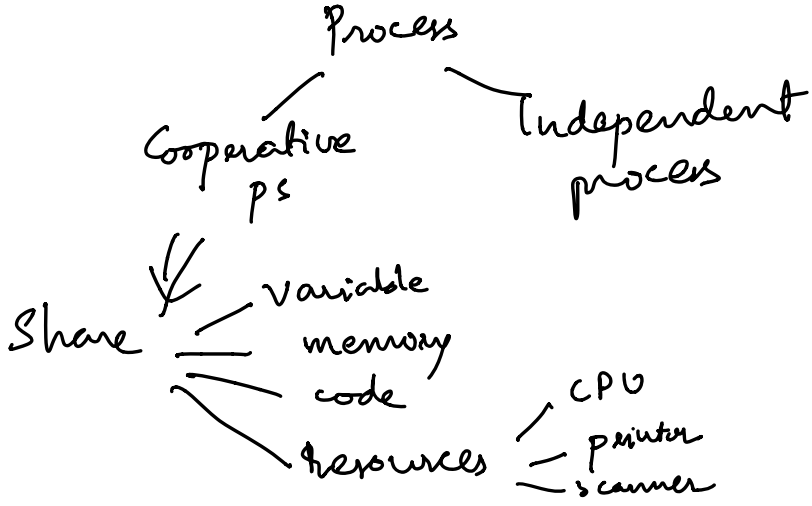
4) Round Robin → criteria: time-quantum
mode: preemptive

context-switching → save running process and bring a new ps (load)

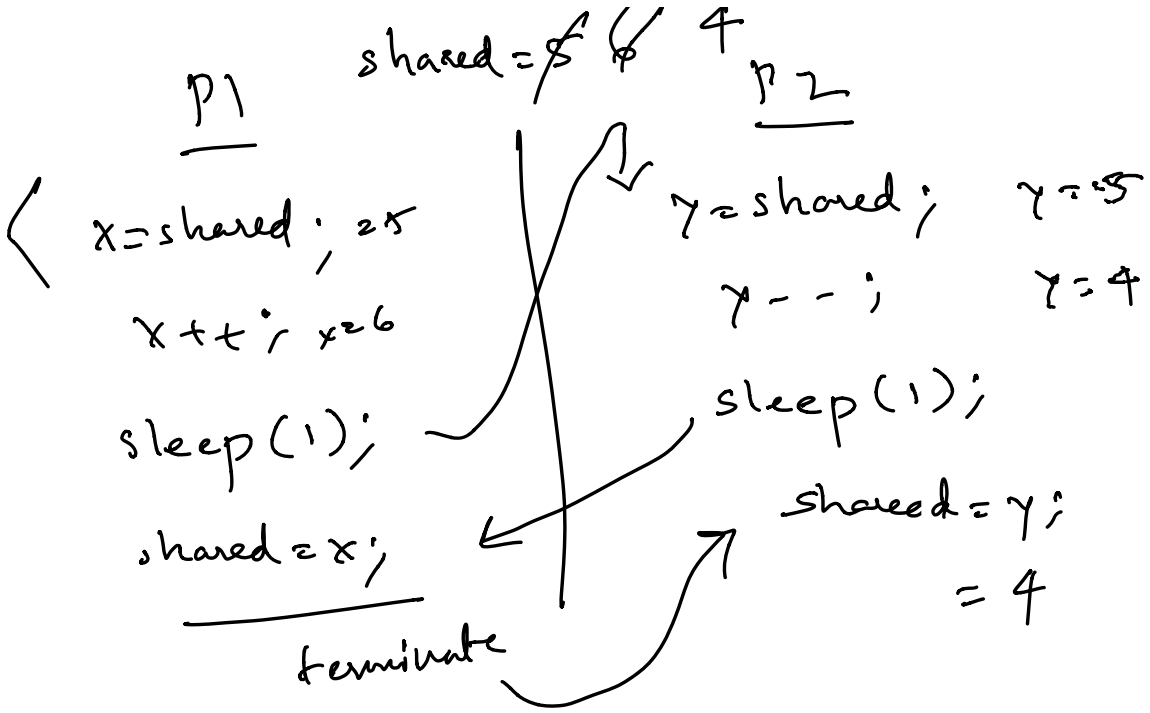
5) pre-emptive priority
→ criteria: "priority"
mode: preemptive

---

(4) Process Synchronization

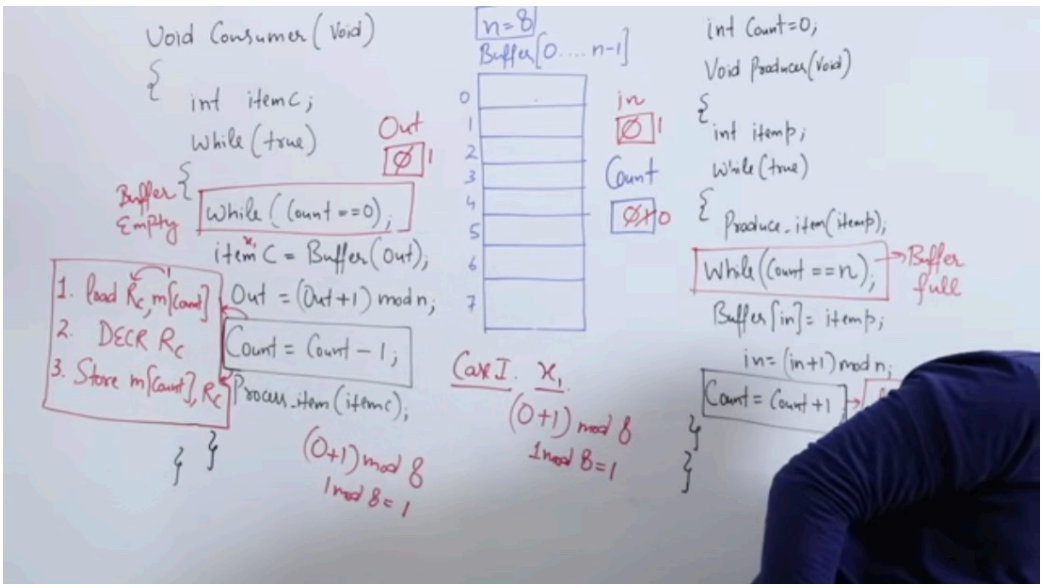


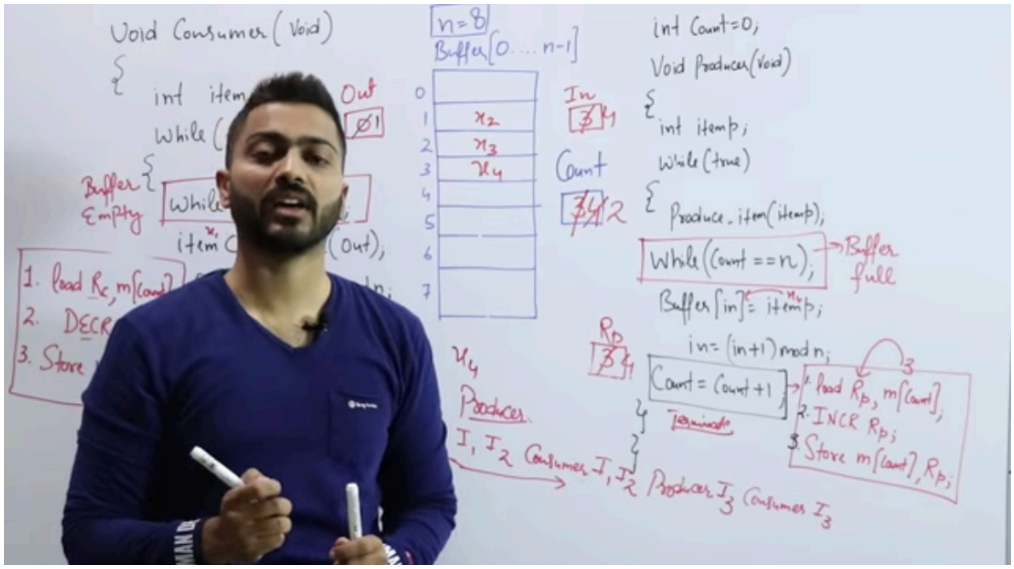
Ex: Race condition

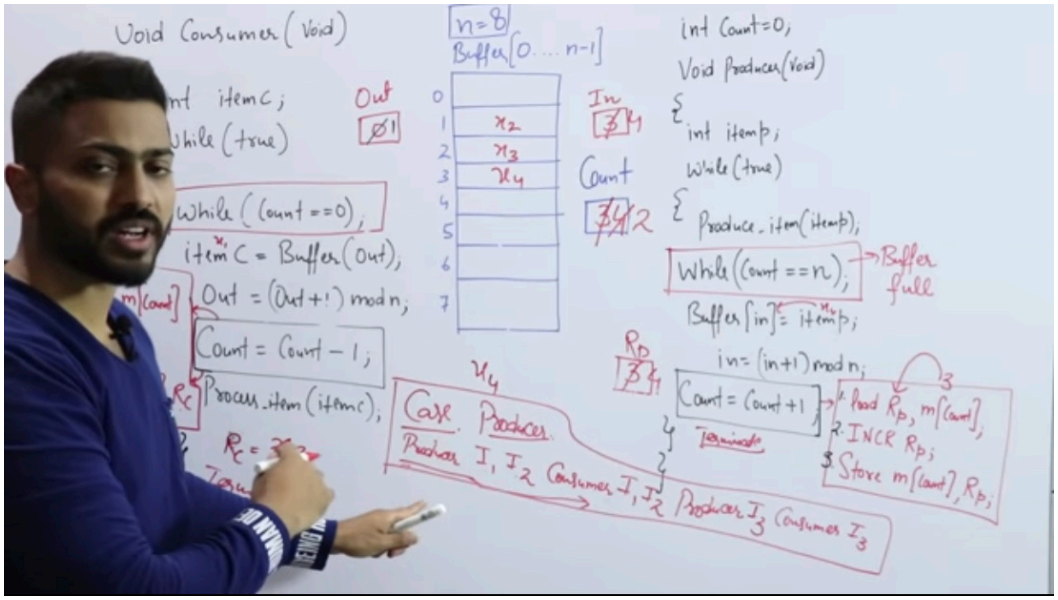


But ans should be 5.

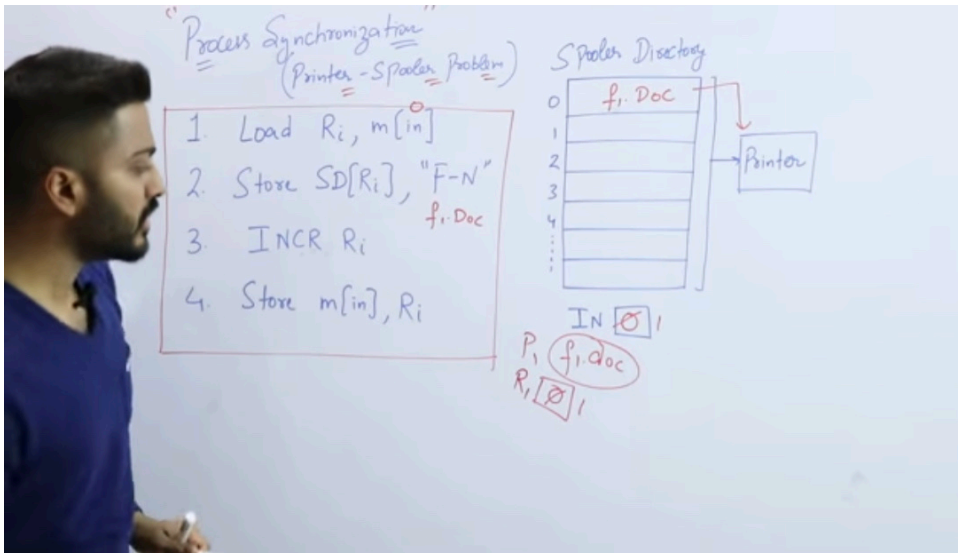
i) Producer – Consumer prob

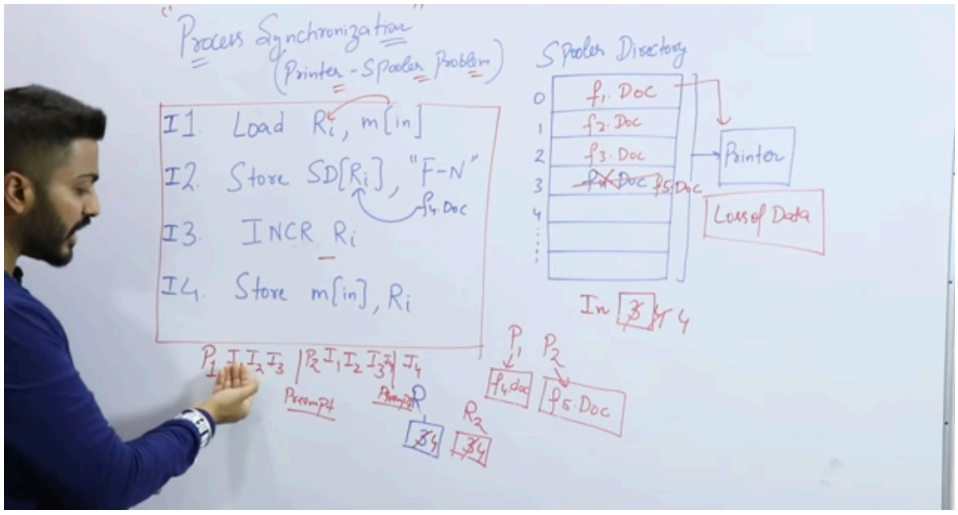



Void Consumer (Void)
n=8
Buffer[0 .... n-1]
int count=0;
Void Producer(Void)
Out
In
Count
Buffer Empty
Buffer full
Produce
Consumer
Terminate
INCR
Store


Void Consumer (Void)
n=8
Buffer[0 .... n-1]
int count=0;
Void Producer(Void)
Out
In
Count
Buffer Empty
Buffer full
Produce
Consumer
Terminate
INCR
Store

ii) Printer - Spooler Problem





iii) CRITICAL SECTION

→ is the part of the program where shared resources are accessed by various processes.

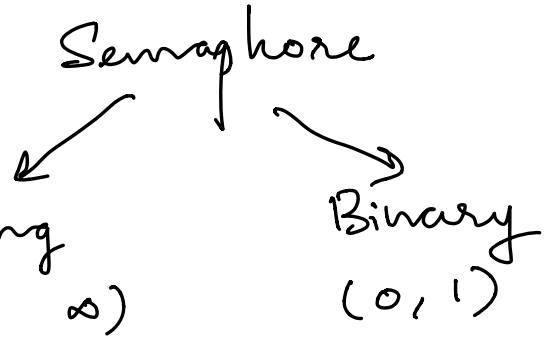
(OR) is a place where shared variables, resources are placed.

Synchronization Mechanism

4 Conditions:

1) Mutual Exclusion } Primary / Mandatory
2) Progress
3) Bounded Wait
4) No assumption related to H/W speed.

(iv) Semaphores → method to prevent race condition & achieve process synchronization (prevent deadlock, loss of data etc).



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

→ Operations : P( ) & V( )

P( ) ↙ 1) down, 2) wait (entry)

V( ) ↙ 1) up, 2) signal, post, release. (Exit)

```
Wait ( Semaphore S)
{   S. Value = S. Value - 1;
    if ( S. Value < 0)
    { put process (PCB) in
      suspend list sleep(),
    }
    else return;
}
Signal (Semaphore S)
{   S. Value = S. Value + 1;
    if (S. Value ≤ 0)
    { Select a process from
      suspend list wake up(); }
}
```

*) Binary Semaphore → 0, 1

```
Wait (Semaphore S)
{ if (S.Value == 1)
    { S.Value = 0; }
  else
    { //Block the ps & put in suspend list
      Sleep();
    }
}
Signal (Semaphore S)
{ if (Suspend list is Empty)
    { S.Value = 1; }
  else
    { //Select a ps from suspend_list
      wake_up();
    }
}
```

Here in wait, signal if "if" section is executed then it's a successful operation

---

→ soln of Producer-Consumer (using Binary Semaphore)

counting semaphore → full = 0
                   → Empty = N
Binary " → S = 1

```
Produce (item p)
{ down (Empty);
  down (S);
  Buffer [IN] = item p;
  IN = (IN+1) % N
  up (S);
  up (full);
}
```

```
Consumer (item c)
{   down (full);
    down (S);
  item c = Buffer [out];
  out = (out+1) % N;
    up (S);
    up (Empty);
}
```

~> soln of reader-writers problem

R_W → problem ; W-R → problem

W-W → problem ; R-R → no prob.

---

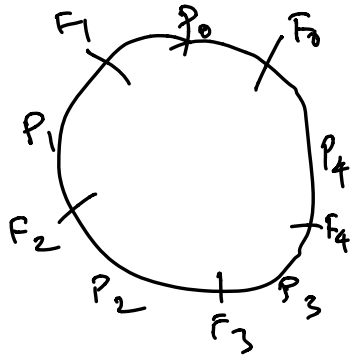
psendo code!

```
int rc = 0
semaphore rS = 1;
semaphore w = 1;
void Reader(){
  while (true)
{   down(rS);
    rc = rc + 1;
    if (rc==1) down(db);
      up(rS);
      [ CS ]
      down(rS);
    rc = rc - 1;
```

```
        if (rc == 0) up(db);
        up(rs);
        process_data
    }
}
void writer() {
    while (true) {
        down(db);
        CS
        up(db);
    }
}
```

⇝ Dining Philosophers

Prob:




```
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);
}
```

If P_0 comes first

say P_0 → fork 0

next P_1 comes & takes fork 1

P_0 has to wait for P_1 to eat and put_fork back. To address this issue we use array of semphorores S[N]; N is no. of forks.

S[5] = {1, 1, 1, 1, 1}

Soln:

```
void philosopher (void)
{
    while (true)
    {
        Think();
        wait (take_fork(Si));
        wait (take_fork(Si+1%N));
        EAT();
        signal (put_fork (Si+1%N));
        signal (put_fork (Si));
    }
}
```

But here, there's a deadlock situation say if every philosophers takes his left spoon & waiting for other spoon.

→ To solve this, just change the orientation for any 1 philosopher

Say $P_0 \rightarrow (S_0)\ S_1$

$P_1 \rightarrow (S_1)\ S_2$

$P_2 \rightarrow (S_2)\ S_3$

$P_3 \rightarrow (S_3)\ S_4$

$P_4 \rightarrow (S_0)\ S_4$

for P_4 reverse the condn, hence P_4 waits for S_0 meanwhile P_3 picks S_4 and eats & so on.

Deadlock will be resolved.

Deadlock

→ If two or more process are waiting on happening of some event, which never happens, then we say these processes are stuck in deadlock,

conditions for deadlock (Necessary)

1) Mutual Exclusion.
2) No preemption.
3) Hold & Wait.
4) Circular wait.

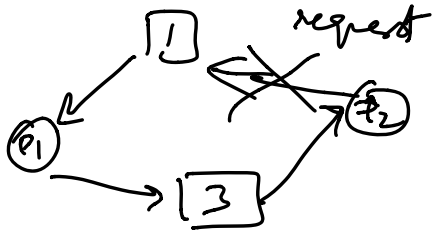
Various methods to handle it

1) Deadlock Ignorance (Ostrich method) — widely in OS used
2) Deadlock prevention
3) Deadlock avoidance (Banker's Algo)
4) Deadlock detection & recovery

① → Ignorance : Cuz of rare occurence of deadlock in OS we ignore it in many OS we use i.e., system hangs & we restart it. If we've extra code it might slow down OS's speed.

② Prevention → try to remove one of the 4 conds of deadlock (eg: to remove circular wait, say that a process can request resource only in the increasing order

i.e. 1. Printer
2. Scanner
3. CPU
4. Register



Banker's Algo (deadlock avoidance)

(Also used for dl. detection)

Ex: Total A=10, B=5, C=7

Ps	Allocation (A)	(B)	(C)	Max Need (A)	(B)	(C)	Rem. Need (A)	(B)	(C)
P1	0	1	0	7	5	3	~~7~~	~~4~~	~~3~~
P2	2	0	0	3	2	2	~~1~~	~~2~~	~~2~~
P3	3	0	2	9	0	2	~~6~~	~~0~~	~~0~~
P4	2	1	1	4	2	2	~~2~~	~~1~~	~~1~~
P5	0	0	2	5	3	3	~~5~~	~~2~~	~~2~~
	7	2	5						

Remaining → 3 3 2

~~P3~~ ↓ P2 ✓

rem → 5 3 2

~~P3~~ ↓ P4 ✓

rem → 7 4 3 → P5 ✓

rem → 7 4 5 → P1 ✓

Sequence

P2 → P4 → P5 → P1 → P3

(Safe sequence)

rem → 7 5 5 → P3 ✓

rem → 10 5 7 = total ✓

If at any situation you can't alloc the resources to any ps then there's a deadlock.

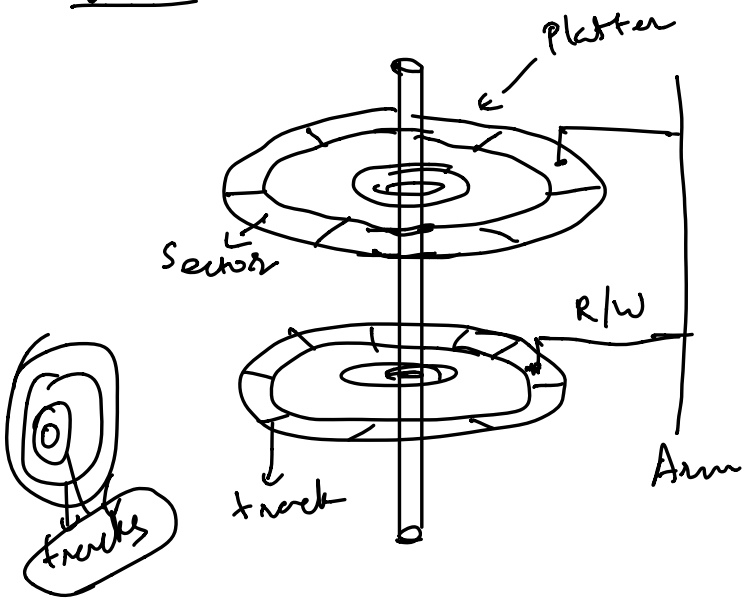
---

(5) DISK SCHEDULING ALGOS

Goal: to minimize the seek time

↓

time taken to reach up to the desired track.



Platter

↓

Surface

↳ track

↓

sector

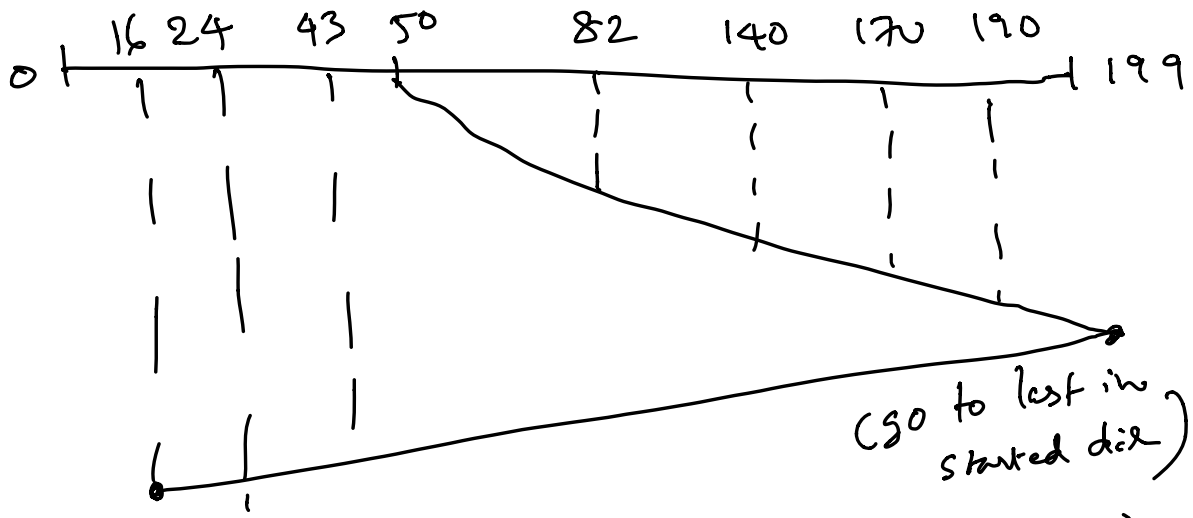
FCFS, SSTF (Shortest seek time first), SCAN, LOOK, CSCAN (circular SCAN), CLOCK.

FCFS ex: disk → 200 tracks, request que → 82, 170, 43, 140, 24, 16, 190

R/w head = 50

total no. of tracks movement by R/w head = (82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16)

SCAN:



moves → (199-50) + (199-16)

In look just go till 190.

In C-SCAN, go from 50 to 199 & then start from '0' again.

moves = (199 − 50) + (199 − 0) + (43 − 0)

In C-look, go from 50 to 190 & then start from lowest value '16'

again

moves = (190 − 50) + (190 − 16) + (43 − 16)

In SSTF, from curr point to next point such that seek time is min ⇒ greedy (prob is starvation)

---

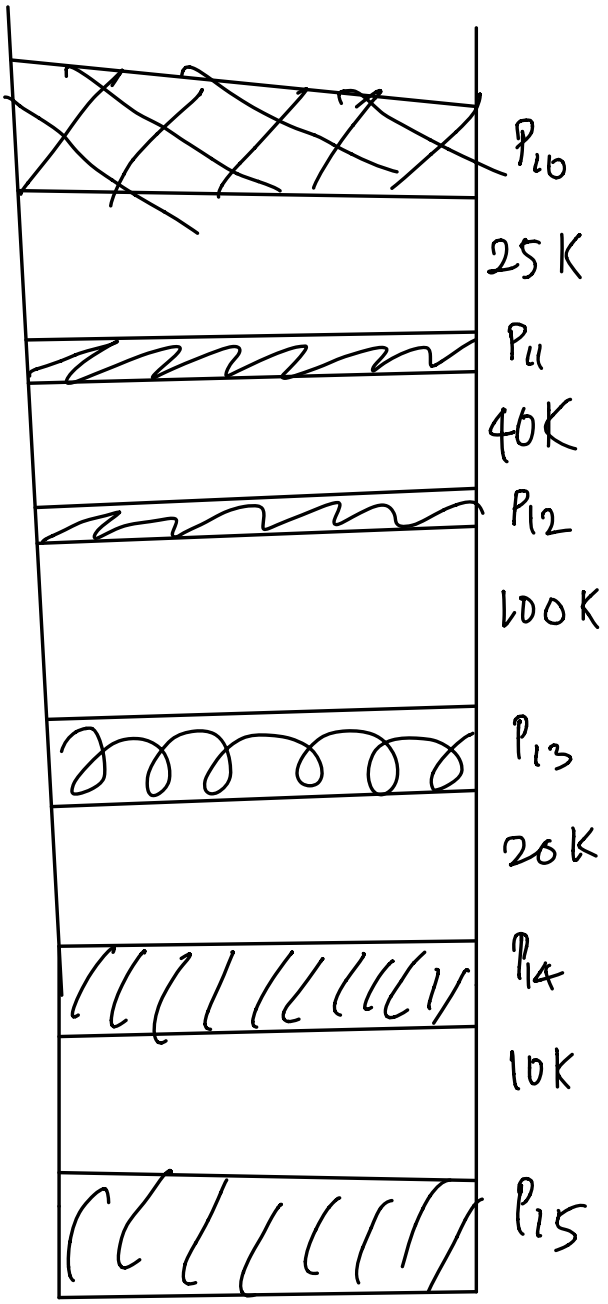
(6) Memory Management

First fit: Allocate the first hole that is big enough

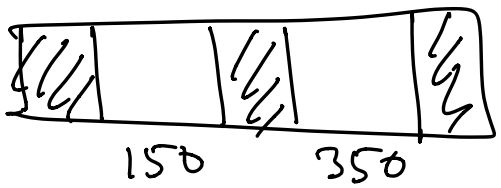
Next fit: Same as first fit but start search always from last allocated hole

Best fit: Allocate the smallest hole that is big enough

Worst fit: Allocate the largest hole.



Ex:



Request from ps → 300k, 25k, 125k, 50k.

the above req could be satisfied by

1) Best & not first

2) Best not & first yes

3) Both

4) None

Sol:

First	Best
300 ↳ 150, 50	300 ↳ 150, 50
25 ↳ 125, 50	25 ↳ 150, 25
125 ↳ 0, 50	125 ↳ 25, 25
50 ↳ 0, 0 ✓	50 ↳ X (External fragmentation)

→ worst fit works too