CAT-2

~~Module~~

Process Synchronization

- Process synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one process try to gain access to the same shared resource or data at the same time.
- This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other process accened the same data. To avoid this type of inconsistency of data, the process need to be synchronized with each other.

Process

Cooperative process

independent process

(5)

6-1=5

4+1=5

context switching

P1

1. int X = shared; X = 5
2. X++; 6
3. sleep(1);
4. shared = X; 6

Process paused for 1 sec.

Terminated.

Shared = 6

P2

int y = shared; y = 5

y--; y = 4

sleep(1);

shared = y;

4

Shared = 4

int shared = 5, Uniprocess.

Now, shared = 6.

Now, shared = 4

Race Condition

6 / 4

P2 P1

This is wrong as the processes are not synchronized

* Race condition :-

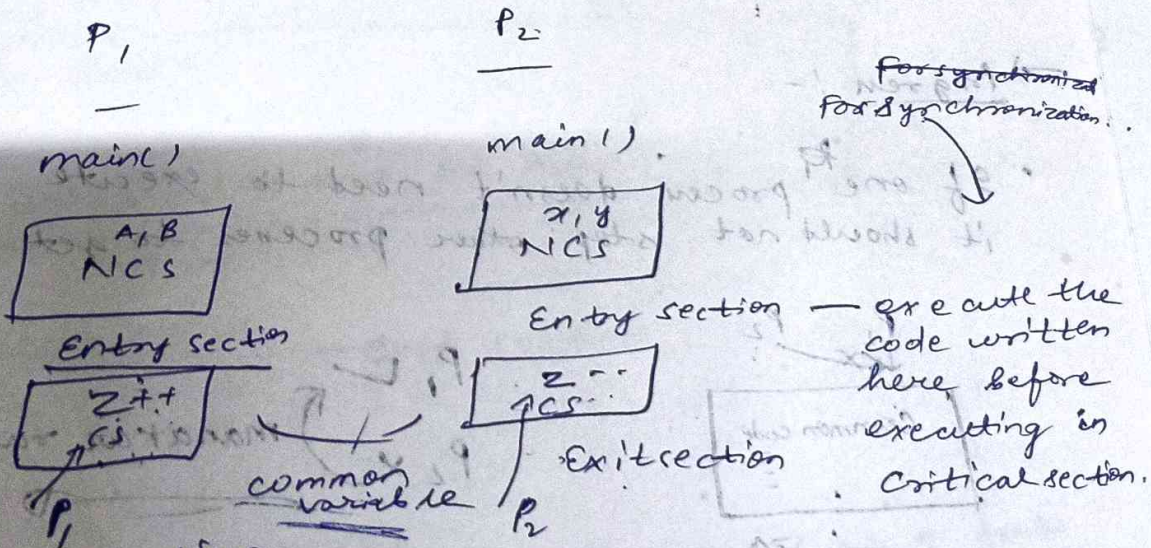
- When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all process try to access and modify this shared resource.
- Thus, all the process race to say that my result is correct. This condition is called the race condition.

* Critical-section Problem :-

- It is part of the program where shared resources are accessed by various process.

↓ cooperative (here).

→ Critical section - place where shared variables, resources are placed.

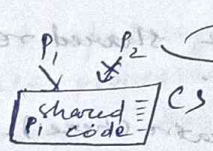


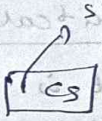
* Synchronization mechanism :-

4 conditions

1) Mutual Exclusion } Primary
2) Progress }
3) Bounded Wait } Secondary
4) No assumption related to H/w }

Architectural Neutrality

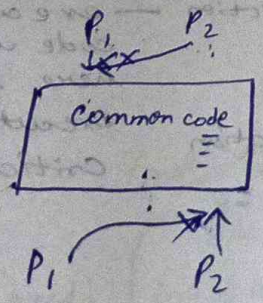


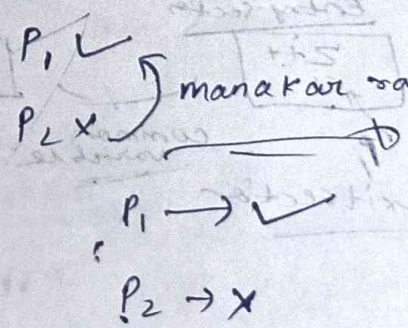


→ Mutual Exclusion :- If one process is executing inside critical section then the other process must not enter in the c.s.

→ Progress :-

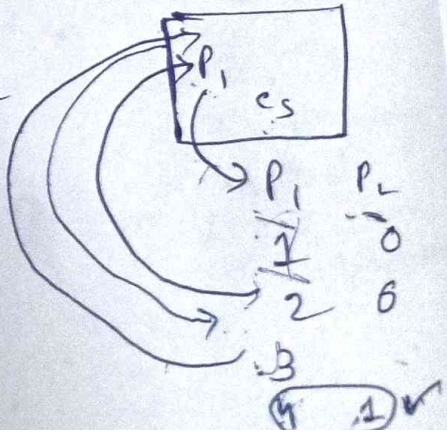
- If one process doesn't need to execute into cs then it should not stop other processes to get into the cs.





→ Bounded Waiting :-

- We should be able to predict the waiting time for every process to get into cs. The process must not be endlessly waiting for getting into the cs.



→ Architectural Neutrality :-

- Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should ~~on~~ also run on other ones as well.

→ Solutions for critical section Problem :-

(1) Peterson's solution :-

- must have only 2 process $\{P_0, P_1\}$

data items { int turn;
boolean flag [2]; → initial = False → both P_0 and P_1 are not interested to enter the CS.

→ Structure of procn P_i :-

```
while (true)
{
    flag[i] = true;
    turn = j;
    while (flag[j] == true && turn == j);

    Critical section

    flag[i] = false;
}
```

entry section

exit section

Structure of procn P_0 :-

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1);

    C.S.

flag[0] = false;
```

// flag[j] = true
turn = i;

→ Structure of process P_1 :-

P_0 P_1
P_i P_j

```
flag [i] = true;
turn = 0;
while (flag [0] == true && turn = 0);
                C.S.
flag [i] = false;
```

- To prove that this method is a solution for the C.S. problem, we need to show: Mutual exclusion is preserved.

→ P_i enters its C.S. only if either

flag [j] == false or turn == i.

→ If both processes want to enter their critical sections at the same time,

then flag [i] == flag [j] == true.

→ However, the value of turn can be either 0 or 1 but cannot be both.

Hence, one of the processes must have successfully executed the while statement (to enter its C.S.), and the other process has to wait, till the process leaves its C.S.

- Mutual exclusion is preserved.

For bounded waiting Progress requirement :-

- Case - 1 :-

→ P_i is ready to enter its C.S.

→ If P_j is not ready ; then flag[j] == false, and P_i can enter its C.S.

- Case - 2 :-

→ P_i and P_j are both ready to enter its C.S.

flag[i] == flag[j] == true.

Either turn == i or turn == j

If turn == i, then P_i will enter the C.S.

If turn == j, then P_j will enter the C.S.

~~Thus, the bounded wai~~

Bounded - waiting :-

→ Once P_j exits its C.S., it will reset flag[j] to false, allowing P_i to enter its C.S.

→ Even if P_j immediately resets flag[j] to true, it must also set turn to i.

→ Then P_i will enter the C.S. after at most one entry by P_j.

(2) Synchronization Hardware :-

```
do {
    acquire lock
        CS
    release lock
}
```

- executes in user mode.
- Multi-process solution.
- ~~Sto~~ mutual exclusion.
- ~~Ma~~ Guarantee →

Normal lock → ME guarantee

Structure :-

```
1. while (lock == 1);
2. lock = 1;
```
Entry code

3. Critical section

```
4. lock = 0
```
EXIT CODE

lock = 0 → vacant CS
1 → Process

Case 1 :-

Initial lock = 0

lock = 0 → vacant CS
1 → false

Initial lock = 0 / 1

Case-2 :-

P_1	P_2	$L = 0 \rightarrow 1$
1*	1 2 3 (CS)	P2 (CS)

* Test and set Synchronization Hardware :-

→ To avoid preemption

```
while (test_and_set (&lock));
```

false → P_1 ; true → P_2 ✗ → ME

```
while (clock == 1);
lock = 1
```

CS

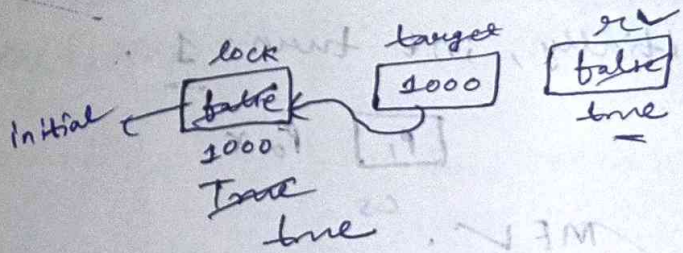
```
lock = false;
```

```
Boolean TestAndSet (Boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

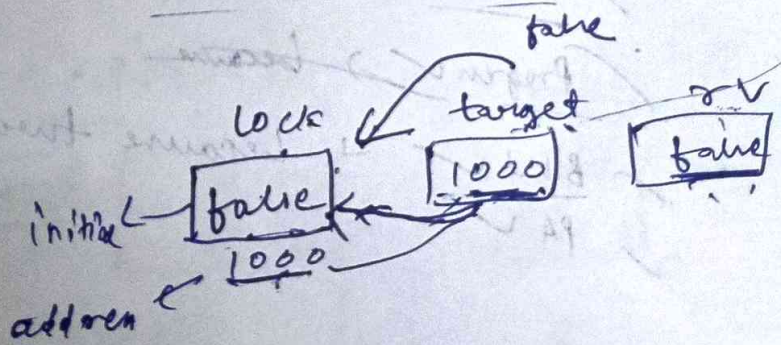
false / true

false 0, 1

true



ME ✓



★ Turn Variable (Strict Alteration) :-

- 2 process solution.
- Run in user mode.

Process "P_0"	Process "P_1"
entry code → while (turn == 0);	while (turn != 1);
CS	CS
exit code → turn = 1;	turn = 0;

- If initially, int turn = 0.

P_0 — CS

P_1 ✗ — Because false. if true then it is stuck within and P_2 cannot enter CS.

- If initially, int turn = 1.

P_1 — CS

P_0 ✗

ME ✓

Progress ✓ → ~~because~~

BW ✓ → because turn is used.

PA ✓

(3) Semaphores :

Counting (-∞ to +∞) | Binary (0,1)

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

Wait, signal.

entry [P(), Down, Wait]

exit [V(), Up, Signal, Post]

entry section code

exit section code

```
Down( semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0)
    {
        Put procon (PCB) in suspended list, sleep();
    }
    else
        return;
}
```

```
Up(semaphore s)
{
    s.value = s.value + 1;
    if (s.value <= 0)
    {
        Select a process from suspended list
        wake up();
    }
}
```

S=S-1

S: 1 → 0 → -1 → -2

P1, P2, P3, P4, P5 entry section

CS: P1, P2, P3

P4, P5 → blocked list

Don't put the process in CS.

block P4,

P4 → blocked list

Ex - S: -4 → 4 processes already in blocked list.

continue.

S: -2 → -1 → 0

P4, P5 → P5 → P4

• Classic Definition of Wait and Signal :-

```
wait(s)
{
  while s<=0 do noop;      /* busy wait! */
  s = s-1;                 /* s>=0 */
}
signal(s)
{
  s = s+1;
}
```

Q) S [10]

How many process can succesfully enter into the CS?

Ans - 10.

Q) S [10], If we perform 6P operations and 4V operations then what will be the semaphore value?

Ans - 10 9 8 7 6 5 4 10-6=4

10 - 6 = 4

4+4=8.

Q) S [17], 5P, 3V, 1P

17 - 5 = 12

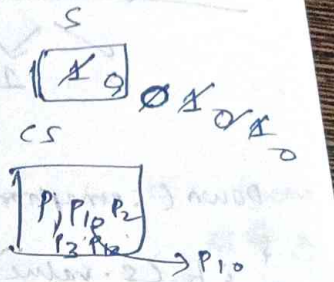
12 + 3 = 15

Q) S [7] · 20 P , 15 V.

7 - 20 = −13

−13 + 15 = 2 .

[S = 2]



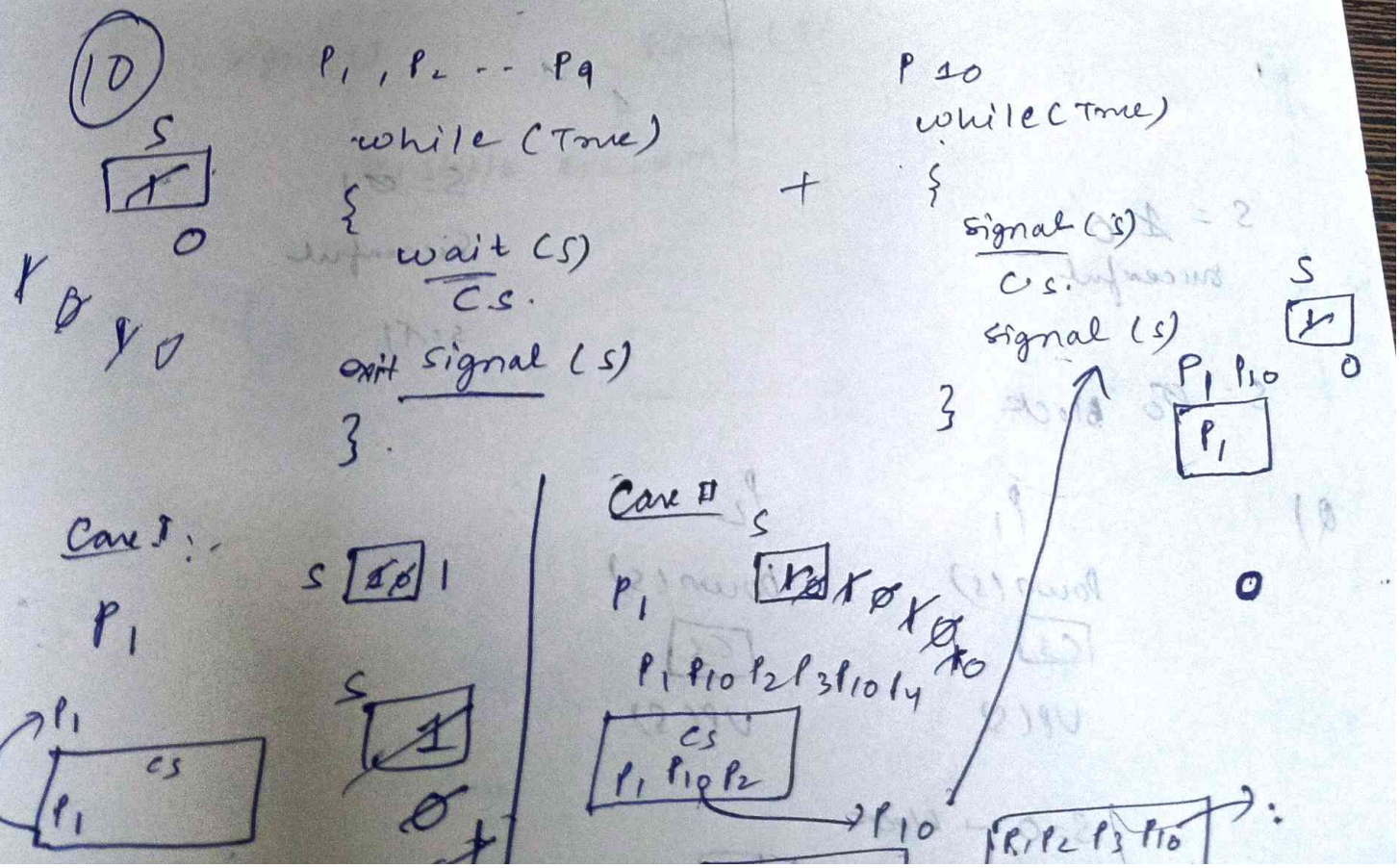
Binary semaphore [0, 1]

Q) S [1]

Consider 10 processes P1, P2 --- P10.

All processes have same code as given below but, P10 has signal (s) in place of wait(s).

If all processes to be executed only once, then maximum number of processes which can be in critical section together?



* Binary semaphore :-

```
Down (semaphore s)
{
  if (s.value == 1)
  {
    s.value = 0;
  }
  else
  {
    Block this procen and place in suspend list, sleep();
  }
}
```

```
Up (semaphore s)
{
  if (suspend list is empty)
  {
    s.value = 1;
  }
  else
  {
    select a procen from suspend list and wake up();
  }
}
```

S = 1 0
succesful

S = 0 0 block

S = 0 1
succesful

S = 1 1

Q)

P_1	P_2
Down(s)	Down(s)
CS	CS
UP(s)	UP(s)

S = 0 → block

P1 to P9 P10

Q) wait ( ) signal ( )

CS + CS

signal ( ) wait ( )

max. no. of process which can be seen in C.S. together

P1, P10, P2 = 3

CS
P1 P10 P2 → maximum = 3

Q) wait (s) signal (s)

signal ()

signal (s)

multiple execution

★ Clanical problems of synchronization :-

(1) Bound - Buffer Problem / Producer - Consumer Problem

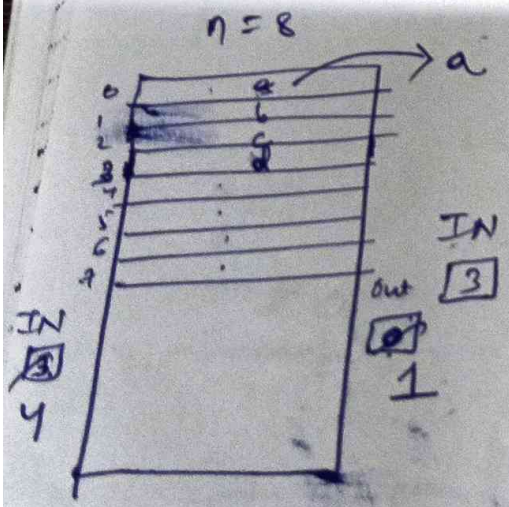
Counting Semaphore → full = 0 → No. of filled slots.
→ empty = N → No. of empty slots.

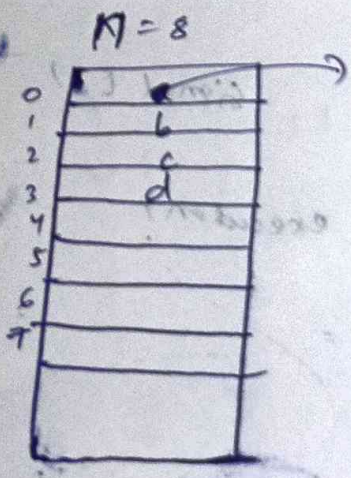
```
Produce-item (itemp);
1) down(empty);
2) down(s);
   buffer[IN] = itemp;
   In = (In+1) mod n;
3) UP(s);
4) UP(full);
```

Exit code

Consume

```
1) down(full);
2) down(s);
   itemc = Buffer[out];
   out = (out+1) mod n;
3) UP(s)
4) Up(empty);
```





0+1 mod 8 = 1

Empty = 8 ↓ 5
full = 3 ↑ 4
In = 4 mod 8
In = 4

Emp ↓ 8 ↓ 5
full = 3 ↑ 3

s = 1 ↓ 0 ↑

In = (3+1) mod 8
In = 4

1 ↓ 0 ↑ 1

mod 8

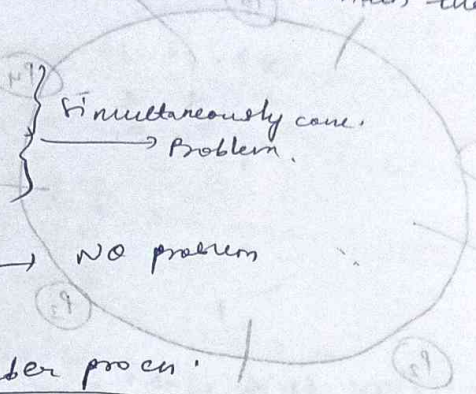
(2) Readers - Writers Problem

- No. of writer and reader must remain the same.

Same data

R - W
W - R
W - W
} Simultaneously come. → Problem.

R - R → No problem



Structure of reader proc.

```
int do {
    wait (mutex);
    read count ++;
    if (read count == 1)
        wait (wrt);
    signal (mutex);
        // reading is performed
    wait (mutex);
    read count --;
    if (read count == 0)
        signal (wrt);
    signal (mutex);
} while (TRUE);
```

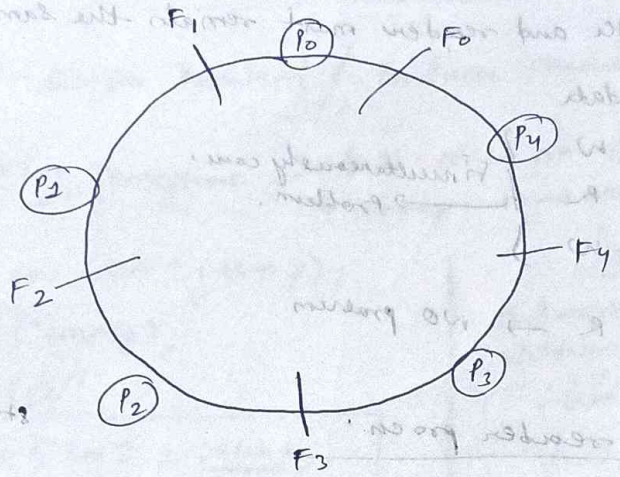
Str of writer

```
do{
    wait (wrt);
    // writing is performed.
    signal (wrt);
}
while (TRUE);
```

Case - I

mutex [1] RC [0] wrt [1]

★ Dining Philosophers Problem :-



5 philosophers
5 chopsticks

S_0 S_1 S_2 S_3 S_4 [Five semaphores]

Structure :-

```
while (true) {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);

    /* eat for a while */

    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);

    /* think for a while */

}
```

P0 : S_0 S_1

P1 : S_1 S_2

P2 : S_2 S_3

P3 : S_3 S_4

P4 : S_4 S_0

- Two philosophers can come together only if they are using different chopsticks. Like, ~~S_0 and S_1, S_2 and~~ P_0 and P_2.

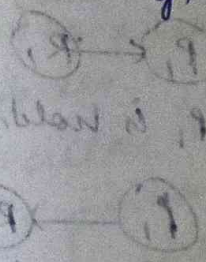
- If any case, S_0 S_1 S_2 S_3 S_4
  1 1 1 1 1

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

→ only first wait() ~~too~~ executes and then it gets preempt to another procm then the situation will be a deadlock situation as all the semaphores value becomes 0 and it remains zero.

S_0, ~~just~~ to ~~set~~ unlock the deadlock, we need to give P_4 : S_0 S_4 → reverse

so, that S_4 remains one and can reach to signal



★ Deadlock :-

- When two or more concurrent process are prevented from completing their task, it is known as a deadlock situation.

Deadlock can occur if 4 conditions hold simultaneously.

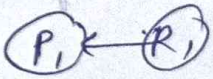
(1) Mutual exclusion :- only one process at a time can use a resource.

(2) Hold and wait :- a process holding at least one resource is waiting to acquire additional resources held by other processes.

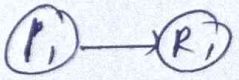
~~(3) No preemption :- a resource can be released only voluntarily by the process holding it, after that process.~~

(3) No preemption :- A resource can be released only voluntarily by the process holding it, after the process has completed its task.

4) Circular wait :- forms a loop.

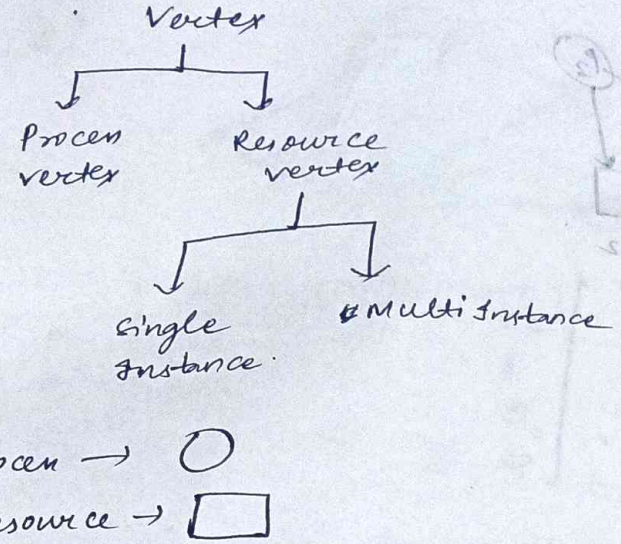


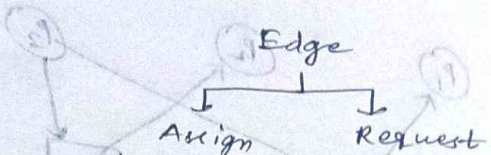
P_1 is holding R_1

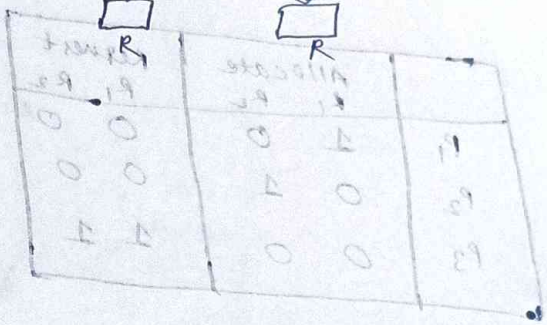


P_1 needs R_1.

★ Resource – Allocation Graph :- (RAG)



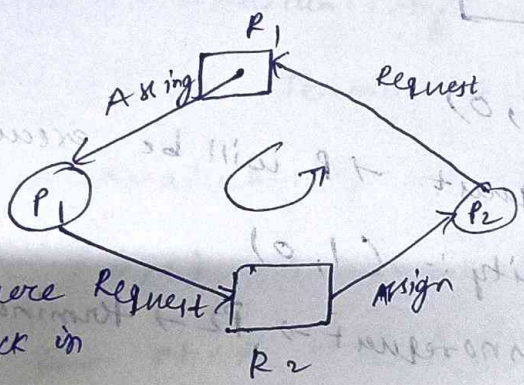




Process → ○

Resource → □

Q)



To check if there will be a deadlock in this RAG :-

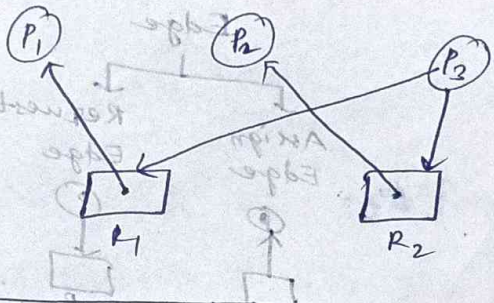
- Circular Wait ↪

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

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

So, There is a deadlock situation.

a)



	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

Availability :- (0, 0)

As, P_1 has (0,0) + request → P_1 will be executed and terminate.

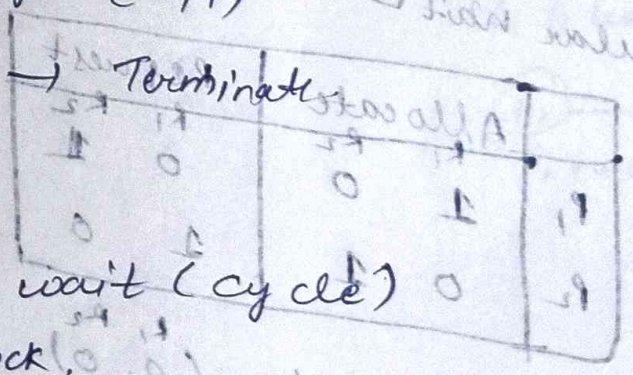
Now, new availability :- (1, 0)

Now, P_2 has no request → P_2 → terminate.

Now availability (1,1)

P_3 → (1,1) → Terminate

No deadlock.

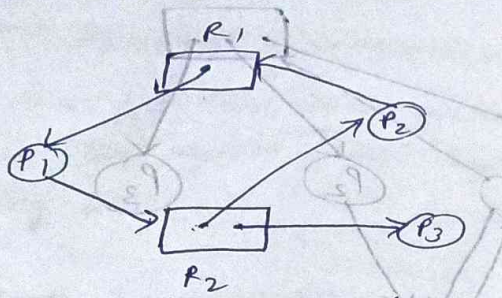


Single instance :-

- If RAG has circular wait (cycle)
  ⇒ Always Deadlock.
- If RAG has no cycle, then no deadlock.

→ Multi-Instance RAG :-

Q)



	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 Availability: $(0,0)$ (R_1, R_2)

P_3 (0,0) → Terminated.

$$\begin{array}{r} 00 \\ +\,01 \\ \hline 01 \end{array}$$

Normal ~~binary~~ Just add.

New Availability → (0, 1)

P_1 (0,1) → Terminated.

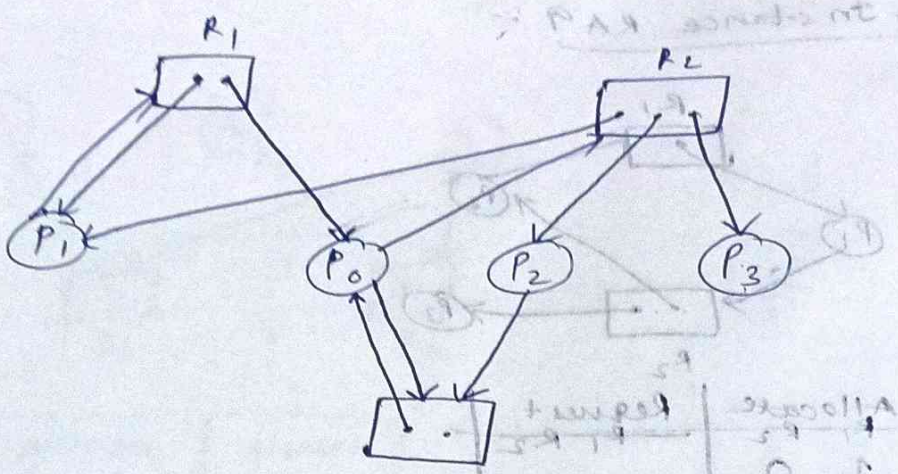
$$\begin{array}{r} 01 \\ +\,10 \\ \hline 11 \end{array}$$

~~No~~ N.A → 1 1

P_2 ←→ Terminated.

So, No deadlock even if there is a cycle, there's no deadlock as it is multi-instance.

Q)



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

Current Availability : $(R_1\ R_2\ R_3)$ (0 0 1)

$P_2 \rightarrow$ Terminated.

0 0 1
+ 0 1 0
N.A → 0 1 1

$P_0 \rightarrow$ Terminated.

0 1 1
1 0 1
NA → 1 1 2

$P_1 \rightarrow$ Terminated

1 1 2
+ 1 1 0
NA → 2 2 2

$P_3 \rightarrow$ Terminated.

222

No deadlock

* Deadlock Handling Methods :-

1) Deadlock Ignorance (ostrich Method)

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

2) Deadlock Prevention :

Restrain the ways request can be made.

→ Mutual Exclusion :- not required for sharable resources (eg: read-only files); must hold for non-sharable resources.

→ No preemption :-

- If a process that is holding some resources requests another resource that can't be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.

→ Hold and Wait :-

- must guarantee that whenever a process requests a resource, it does not hold any other resources.

→ Circular Wait :-

- impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

3) Deadlock Avoidance (Banker's Algorithm) :-

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- System is in a safe state if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of ALL processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- If a system is in safe state ⇒ no deadlock.
- If a system is in unsafe state ⇒ possibility of deadlock.

4) Deadlock detection and Recovery :-

(i) Kill the process

or

(ii) Resource preemption.

★ Banker's Algorithm (Deadlock-Avoidance). Deadlock detection ↓ of deadlock.

Safe → No deadlock.
Unsafe → deadlock.

Total instances = 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_2	2	0	0	3	2	2	5	3	2	1	2	2 ✓
P_3	3	0	2	9	0	2	7	4	3	6	0	0 ✓
P_4	2	1	1	4	2	2	7	4	5	2	1	1 ✓
P_5	0	0	2	5	3	3	7	5	5	5	3	1 ✓
	7	2	3				10	5	7			

10 − 7

Available = Total − Allocation(total)

Remaining Need = Max. Need − Allocation

Check if need is ≤ available.

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

A B C

10 5 7 → same as given.

Safe sequence. No deadlock.

Q) A system is having 3 process each require 2 units of resource 'R'. The minimum no. of units of 'R' such that no deadlock will occur ____.

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

Ans - P_1 P_2 P_3

$3 \times 2 = 6$

↓

resources.

But minimum?

Now, if $R = 2$

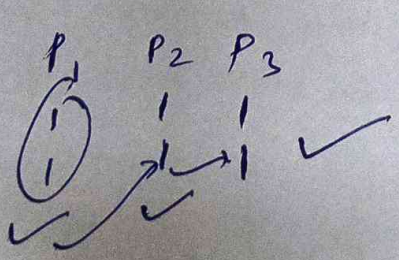
P_1 P_2 P_3

↓ ↓ ↓

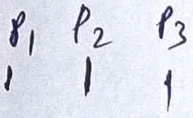
terminate, then P_2 gets the 2, and then P_3 gets.

But 2 from P_1 can go to P_2 (1) and P_3 (1).

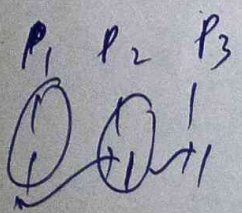
So, again its a deadlock.

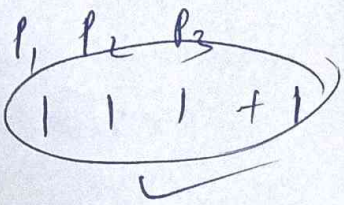


But if



$R = 4$





Ans = 4.

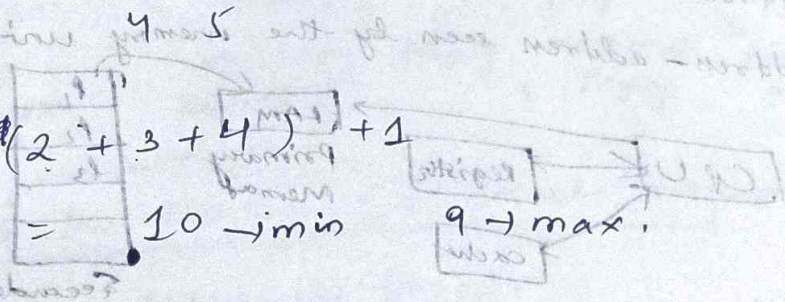
(min req. of each process + 1)

(3+1).

→ max. no. of resources which are allocated but still there is a deadlock?

3. ~

Q) P_1 P_2 P_3



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 _____.

A- R = 4

P_1	P_2	P_3
1	1	1

Virtual Memory

- Separation of user logical memory from physical memory.
- Virtual address space - logical view of how process is stored in memory.
- Virtual memory can be implemented via:
  - → Demand paging
  - → Demand segmentation

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated (v ⇒ in memory - memory resident, i ⇒ not in memory)
- Initially valid-invalid bit is set to i on all entries.
- During MMU address translation, if valid-invalid bit in page table entry is i ⇒ page fault.

→ Page Fault :-

- If there is a reference to a page, first reference to that page will trap to operating system: page fault.

1. Operating System looks at another table to decide:
   - Invalid reference ⇒ abort.
   - Just not in memory.
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory. Set validation bit = v.
5. Restart the instruction that caused the page fault.

★ Demand Paging :-

- Bring a page into memory only when it is needed.
  - → Less I/o needed.
  - → Less memory needed.
  - → Faster response.
  - → More users.

- Page is needed ⇒ reference to it.
  - → invalid reference ⇒ abort
  - → not-in-memory ⇒ bring to memory.

★ Performance of Demand Paging :-

- Page Fault Rate $0 \leq P \leq 1$
  - → if $P = 0$ no page faults.
  - → if $P = 1$, every reference is a fault.

- Effective Access Time (EAT) = $(1-P) \times$ memory access + P (page fault overhead + swap page out + swap page in)

Q) Given,

memory access time = 200 nanoseconds.

Average page-fault service time = 8 milliseconds

$$\text{Av-EAT} = (1-P) \times 200 + P\ (8 \text{ milliseconds})$$

$$= (1-P) \times 200 + P \times 8{,}000{,}000$$

$$= 200 + P \times 7{,}999{,}800$$

** Translation lookaside Buffer (TLB) :-

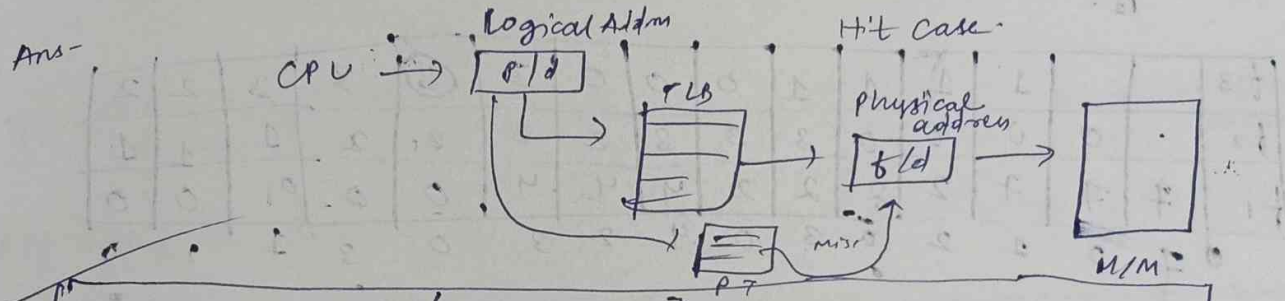
Hit Miss

$$EMAT = (TLB + x) + Miss\ (TLB + x + x)$$

2 mm

Q) A paging scheme using TLB. TLB Access time 10 ns and main memory access time takes 50 ns. What is effective memory access time (in ns) if TLB hit ratio is 90% and there is no page fault.

Ans-



$$EMAT = Hit\ (TLB + M/M) + Miss\ (TLB + PT + M/M)$$

PT (M/M)

PT = M/M

$$= 90\% \cdot (10 + 50) + 10\% \cdot (10 + 50 + 50)$$

$$= 0.9\ (60) + 0.1\ (110)$$

$$= 54 + 11 = 65 \text{ ns}$$

Effective Memory Access Time.

★ Page Replacement :-

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

(1) First-In-First-Out (FIFO) Algorithm :-

Reference string = ~~7,7,1~~ 7,0,1,2,0,3,0,4,2,3,0,3,1,2,0

Pg.

f3			1	1	1	1	0	0	0	3	3	3	3	2	2
f2		0	0	0	0	3	3	3	2	2	2	2	1		1
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
0	7	0	1	2	0	3	0	4	2	3	0	3	1	2	0

- Page Hit = 3
- Page fault / Page miss = 12

$$\text{Hit ratio} = \frac{\text{No. of hits}}{\text{Total No. of reference}} = \frac{3}{15} \times 100 = 20\%$$

$$\text{Miss Ratio} = \frac{\text{No. of miss}}{\text{Total no. of references}} = \frac{12}{15} \times 100 = 80\%$$

→ Belady's Anomaly in FIFO :-

↑
• adds one frame .

↓
It causes more page faults .

eg - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 .

• Normal method (3 frame) : Hits = 3
Pagefaults = 9 .

∴ Belady's Anomaly (4 frames) = Hits = 2
Pagefaults = 10 . ∴

(2) Optimal Algorithm :-
(Replace the page which is not used in longest dimension of time in future)

Ref string → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 .

f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7
	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7

	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1
		0	0	0	0	3	3	3	2	2	2	3	3	3	2	2	2	7
			1	1	1	1	0	0	0	3	3	3	2	2	2	0	0	0
	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7

Page Hits = 11

Page faults = 8

7, 0, 1, 2

3, 0, 4, 2

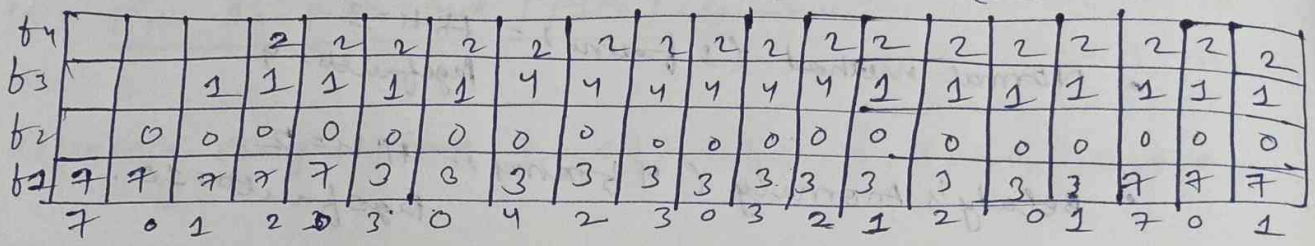
3, 0, 1, 2

7, 0, 1, 2

3) Least Recently Used (LRU)

- Replace the last recently used in past).

Ref → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

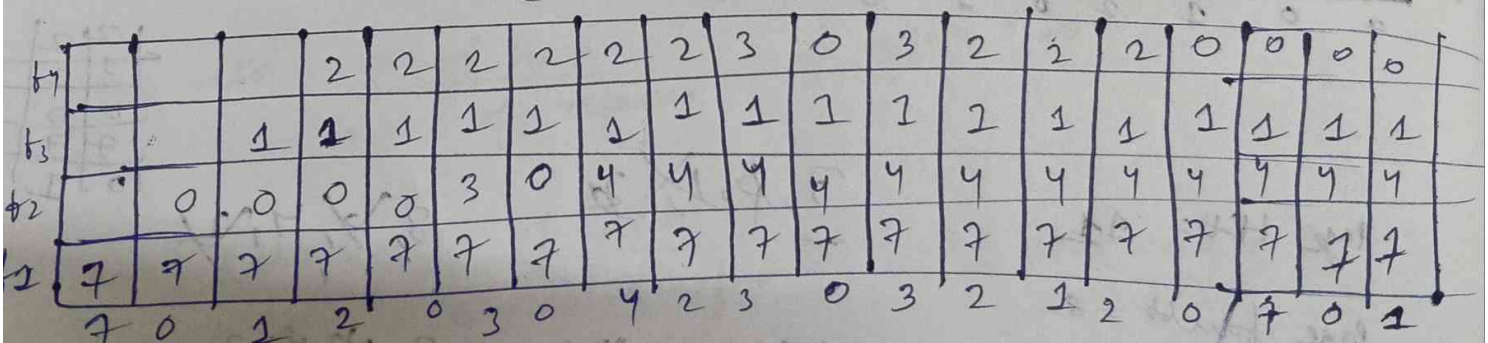


Page hit = 12

Page faults = 8.

→ Most frequently used (MFU) Algorithm :-

Ref → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



* Trashing :-

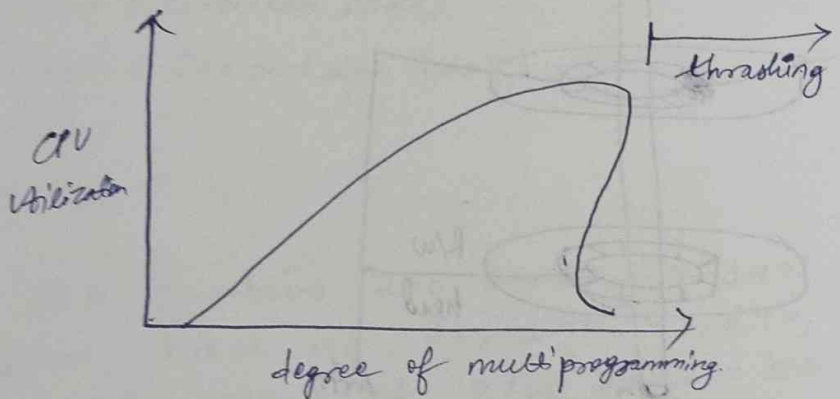
• of a procen does not have "enough" pages, the page-fault rate is very high.

• This leads to:

→ low CPU utilization.

→ Operating System thinking that it needs to increase the degree of multiprogramming

→ Another procen added to the system.



• Why does thrashing occur?

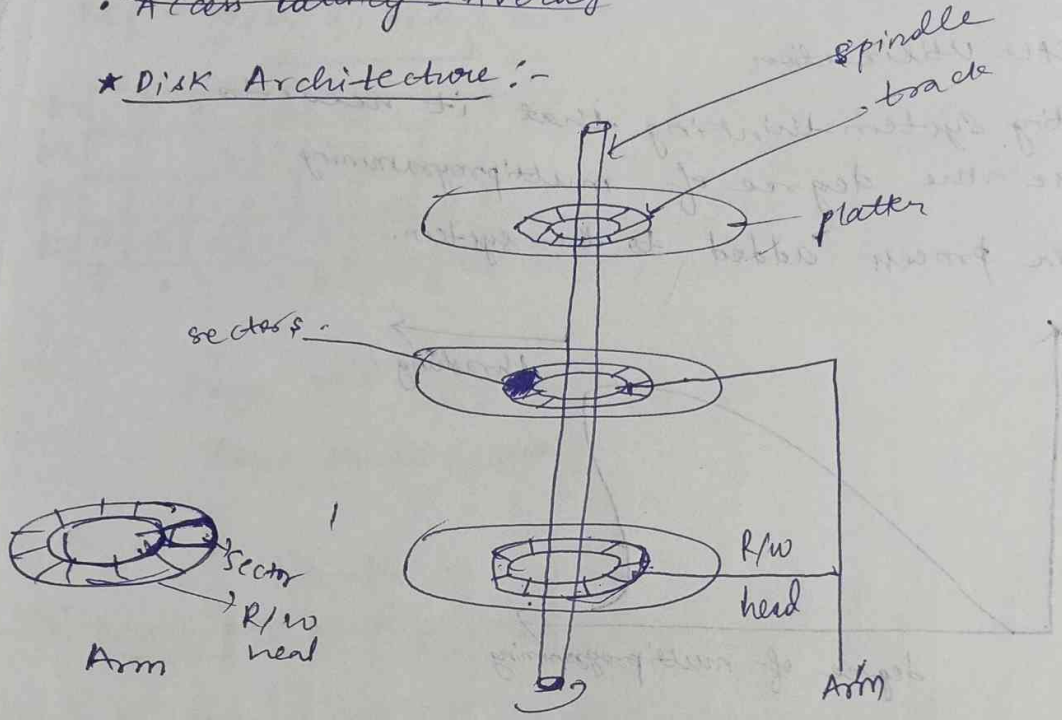
Ans- Σ size of locality > total memory size.

Disks

★ ~~Hard Disk Performance~~ :-

• ~~Access latency = Average~~

★ Disk Architecture :-



★ Disk access time :-

1) Seek Time = Time taken by R/W head to reach desired track.

2) Rotation Time = Time taken for one full Rotation (360°).

3) Rotational latency = Time taken to reach to desired sector (half of rotation time).

4) Transfer Time = $\frac{\text{Data to be Transfer}}{\text{Transfer rate}}$

Transfer rate : No. of Heads x Capacity of one Track x No. of rotations in one second.

Disk Access Time = ST + RT + TT + CT + QT

★ Disk Scheduling Algorithms :-

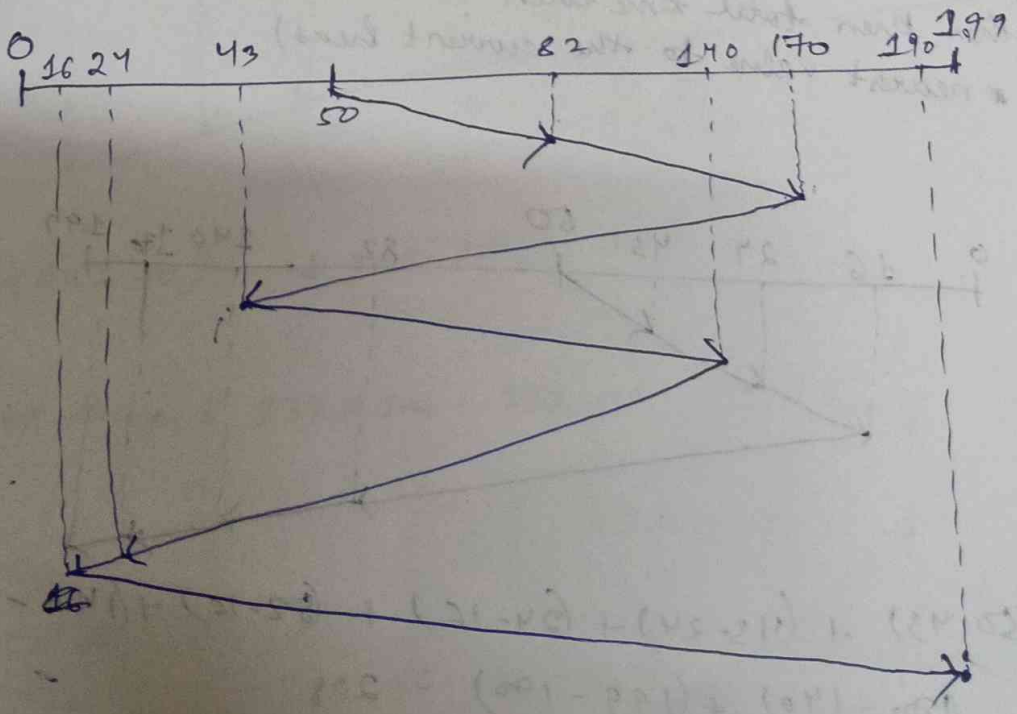
Goal: To minimize the seek time.

Seek Time :- time taken to reach upto desired track.

→ FCFS (First Come First Serve)
→ SSTF (Shortest seek time first)
→ SCAN
→ LOOK
→ CSCAN (Circular scan)
→ CLOOK (Circular look)

(1) FCFS :-

Q) A disk contains 200 tracks (0-199). Request queue contains track no. 82, 170, 43, 140, 24, 16, 190 respectively. Current position of R/W head = 50. Calculate total no. of tracks movement by R/W head.



No. of tracks movement = (82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16) = 642

or

(Total direction) (170-50) + (170-43) + (140-43) + (140-16) + (190-16) = 642

- Advantage = No starvation in this algorithm. (wait)

(2) SSTF :-

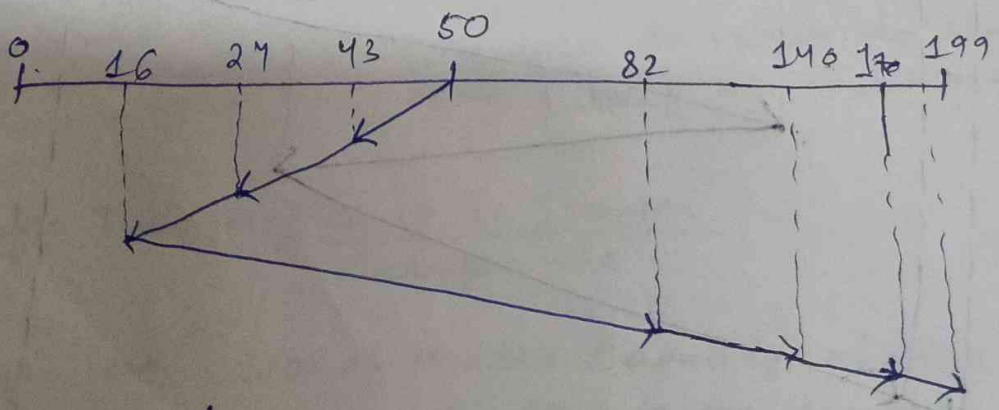
Q) A disk contain 200 tracks (0-199). Request queue contain track no. 82, 170, 43, 140, 24, 16, 190 respectively.

Current position of R/W head = 50

→ Calculate total no. of tracks movement by R/W head using Shortest seek time first?

→ If R/W head takes 1ns to move from one track to another then total time taken?

(* nearest value to the current head)



(50-43) + (43-24) + (24-16) + (82-16) + (140-82) + (170-140) + (199-170) = 208

- Advantage = It tries to give the optimal result.
  - Response time is less

- Disadvantage = Generates overhead (find who is the nearest).

(3) SCAN :- (elevator algorithm)

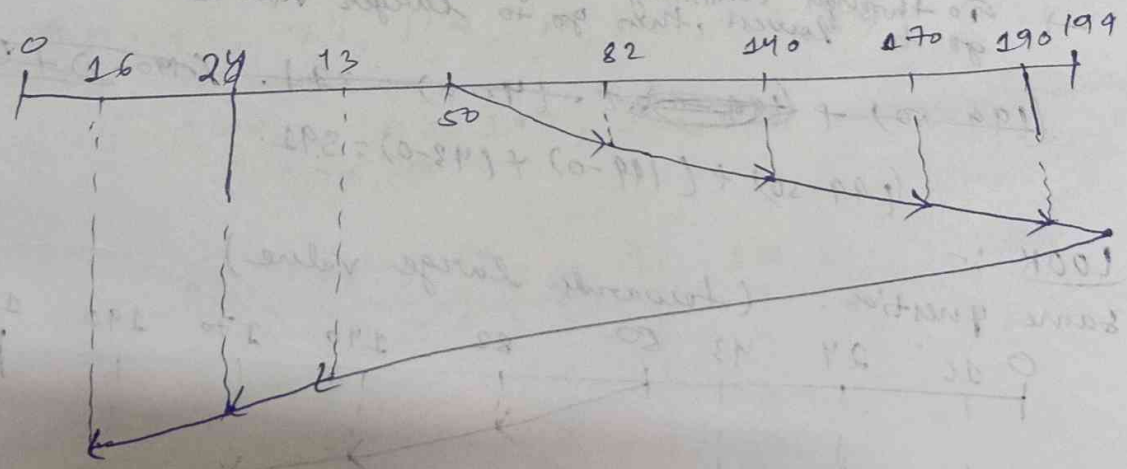
Q) A disk contains 200 tracks (0-199). Request queue contains track no 82, 170, 43, 140, 24, 16, 190 respectively.

Current position of R/W head = 50.

→ Calculate total no. of tracks mores by R/W head using SCAN?

→ If R/W head takes 1ms to move from one track to another then total time taken?

eg - Towards the larger value.



(199-50) + (199-16) = 332

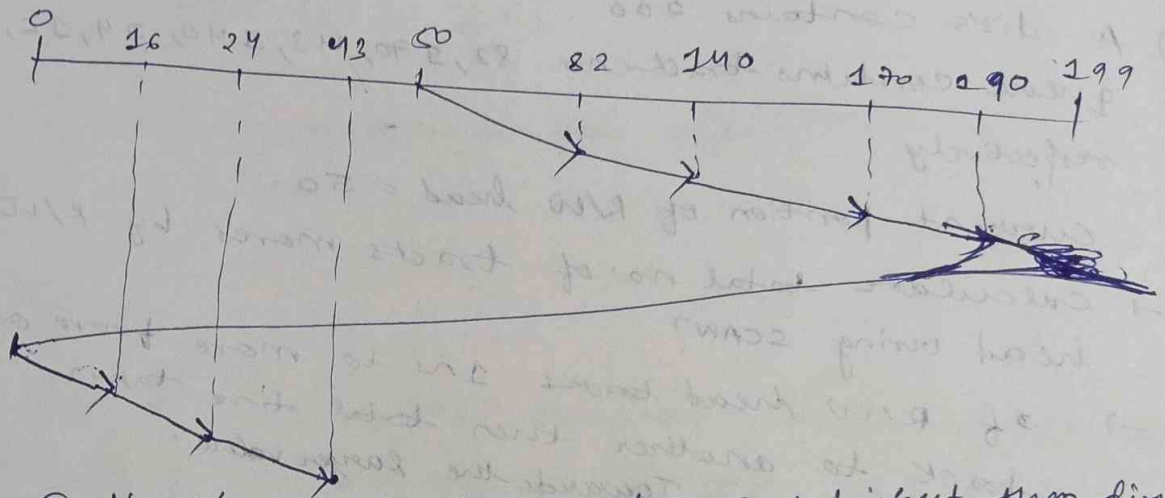
- Total time = 332 × 1ms = 332 ms.

(4) C-SCAN :-

Q) same question as before ?

→ Direction towards larger value.

82, 170, 43, 140, 24, 16, 190



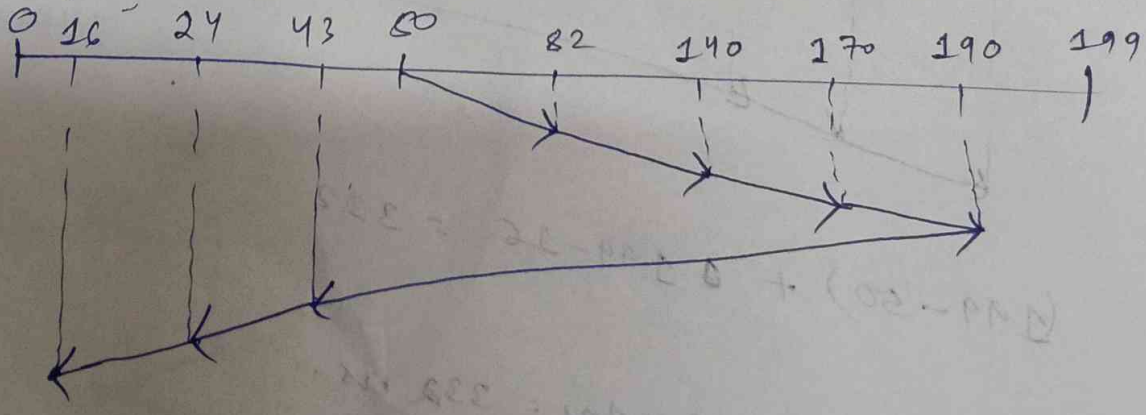
→ Go through scanning larger values, go to highest, then directly go to lowest, then go to larger values.

~~(199-50) + (199-0) + (43-0) = 391.~~ ~~(190-0) + (43-0)~~

(199-50) + (199-0) + (43-0) = 391.

(5) LOOK :-

same question. (towards large value).

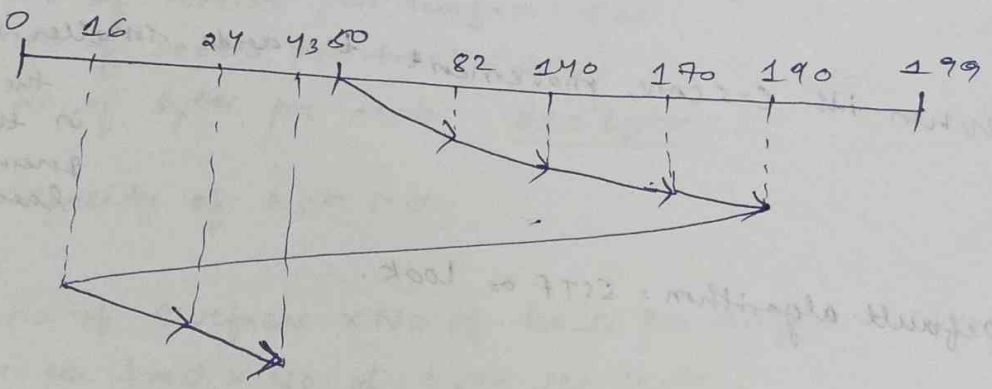


(190-50) + (190-16) = 314.

• Look performs better than SCAN. (Doesn't go extra

(6) C-LOOK :-

Same question. (Towards large value).

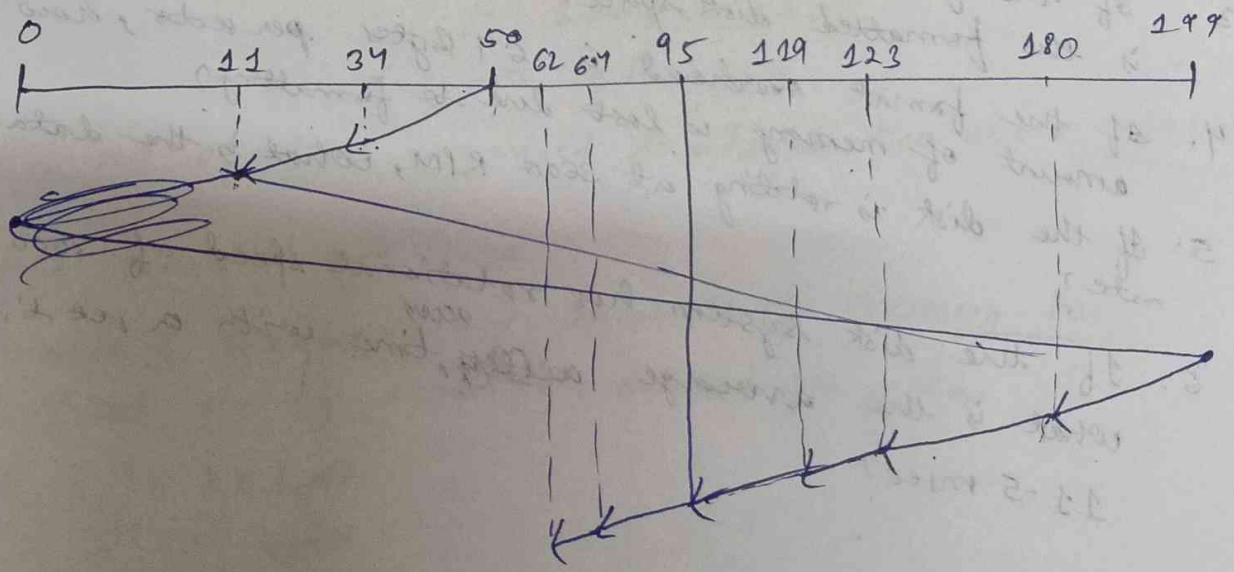


. go to higher values than current value, then come to lowest value then go to higher values.

$(190-50) + (190-16) + (43-16) = 341$

Q) 95, 180, 34, 119, 11, 123, 62, 64

C-scan, 0 to 199, towards smaller track no.
current = 50



$(50-0) + (199-0) + (199-62) =$

$(50-11) + (199-11) + (199-62) =$

- When its C-SCAN, movement towards larger → then go the largest, return to smallest.
- When its C-SCAN, movement towards smaller → then go to the smallest in the ready queue that is the largest end.
- Default algorithm = SSTF or look.

Disk Scheduling Question :-

Q) Consider a disk pack with the following specification.
- 16 surfaces, 128 tracks per surface, 256 sectors per track and 512 bytes per sector.

1. What is the capacity of disk pack?
2. What is the no. of bits required to address the sector?
3. If the format overhead is 32 bytes per sector, what is the formatted disk space?
4. If the format overhead is 64 bytes per sector, how much amount of memory is lost due to formatting?
5. If the disk is rotating at 3600 RPM, what is the data transfer rate?
6. If the disk system has rotational speed of 3000 RPM, what is the average access time with a seek time of 11.5 msec?

Ans — Given,

No. of surfaces = 16
No. of tracks per surface = 128
No. of sectors per track = 256
No. of bytes per sector = 512 bytes.

Part-1 Capacity of Disk Pack.

= Total no. of surfaces × No. of tracks per surface × No. of sectors per track × No. of byte per sector

= 16 × 128 × 256 × 512 bytes
= 2^{28} bytes
= 256 MB

Part-2 No. of bits required to address the sector.

Total no. of sectors = Total no. of surfaces × No. of tracks per surface × No. of sectors per track

= 16 × 128 × 256
= 2^{19} sectors.

∴, No. of bits required to address the sector is 19 bits.

Part-3 Formatted Disk Space.

Formatting overhead → amount of memory lost.

= Total no. of sector × overhead per sector
= $2^{19} \times 32$ bytes
= $2^{19} \times 2^5$ bytes
= 2^{24} bytes = 16 MB.

Now, Formatted Disk Space = Total Disk Space − Formatted overhead
= 256 MB − 16 MB
= 240 MB.

Part 4 : Formatting overhead

Amount of memory lost due to formatting

= Formatting overhead

= Total no. of sectors x overhead per sector

= $2^{19} \times 64$ bytes

= $2^{19} \times 2^6$ bytes

= 2^{25} bytes

Part 5 :- Data Transfer Rate.

No. of rotations in one second

= $\frac{3600}{60}$ rotations/sec

= 60 rotation/sec.

Now, data transfer rate) surface

= No. of heads x capacity of one track x No. of rotation in one second.

= 16 x (256 x 512 bytes) x 60

= $2^4 \times 2^8 \times 2^9 \times 60$ Bytes/sec.

= 60×2^{11} bytes/sec.

Part-6 Average Access Time

Time taken for one full rotation

$= \left(\frac{60}{3000}\right)$ sec

$= 0.02$ sec

$= 20$ msec;

Average rotational delay

$= (1/2) \times$ Time taken for one full rotation

$= \frac{1}{2} \times 20$

$= 10$ msec.

Now, average access time

= Average seek time + Average rotational delay + other factors

$= 11.5$ msec $+ 10$ msec $+ 0$

$= 21.5$ msec.

★ RAID Structure :-

- RAID - multiple disk drives provides reliability via redundancy.
- RAID is arranged into six different levels.
- RAID schemes improve performance and improve the reliability of the storage system by storing ~~redundata da~~ redundant data.

→ Mirroring or shadowing keeps duplicate of each disk.

→ Block interleaved parity uses much less redundancy.

- RAID levels.

(a) RAID 0 : non-redundant striping
(b) RAID 1 : mirrored disks
(c) RAID 2 : memory-style error-correcting codes.
(d) RAID 3 : bit-interleaved Parity
(e) RAID 4 : block-interleaved Parity.
(f) RAID 5 : block-interleaved distributed parity.
(g) RAID 6 : P+Q redundancy.

★ Disk Attachment :-

- Disk may be attached one of two ways:

1. Host attached via an I/O port.
2. Network attached via a network connection.

File-System Interface

★ File Concept :-

- Contiguous logical address space.
- Types :
  - → Data :- Numeric, character, Binary.
  - → Program

★ File Attributes :-

- Name - only information kept in human-readable form
- Identifier - unique tag (number) identifies file within file system.
- Type - needed for systems that support different types.
- Location - pointer to file location on device.
- Size - current file size
- Protection - controls who can do reading, writing, executing
- Time, date and user identification - data for protection, security and usage monitoring.
- Information about files are kept in the directory structure, which is maintained on the disk.
- Information kept in the directory structure :

★ File operations :-

- File is an abstract data type.
- Create
- Write - at write pointer location
- Read - at read pointer location
- Reposition within file - seek
- Delete
- Truncate
- Open (F_i) - search the directory structure on disk for entry F_i and move the content of entry to memory.
- Close (F_j) - move the content of entry F_i in mem

→ Open Files :-

- Open - file table : tracks open files.
- File - pointer : pointer to last read / write location, per process that has the file open.
- File - Open Count : Counter of number of times a file is open - to allow removal of data from open - file table when last process closes it.
- Access rights : per - process access mode information.

→ Open File locking :-

- Shared lock - similar to reader lock - several processes can acquire concurrently.
- Exclusive lock - similar to writer lock.

→ Mediates access to a file :-

- Mandatory - access is denied depending on locks held and requested.
- Advisory - processes can find status of locks and decide what to do.

★ File Structure :-

- None - sequence of words, bytes
- Simple record structure
  - → Lines
  - → Fixed length
  - → Variable length
- Complex structures
  - → Formatted document
  - → Relocatable load file

★ Access Methods :-

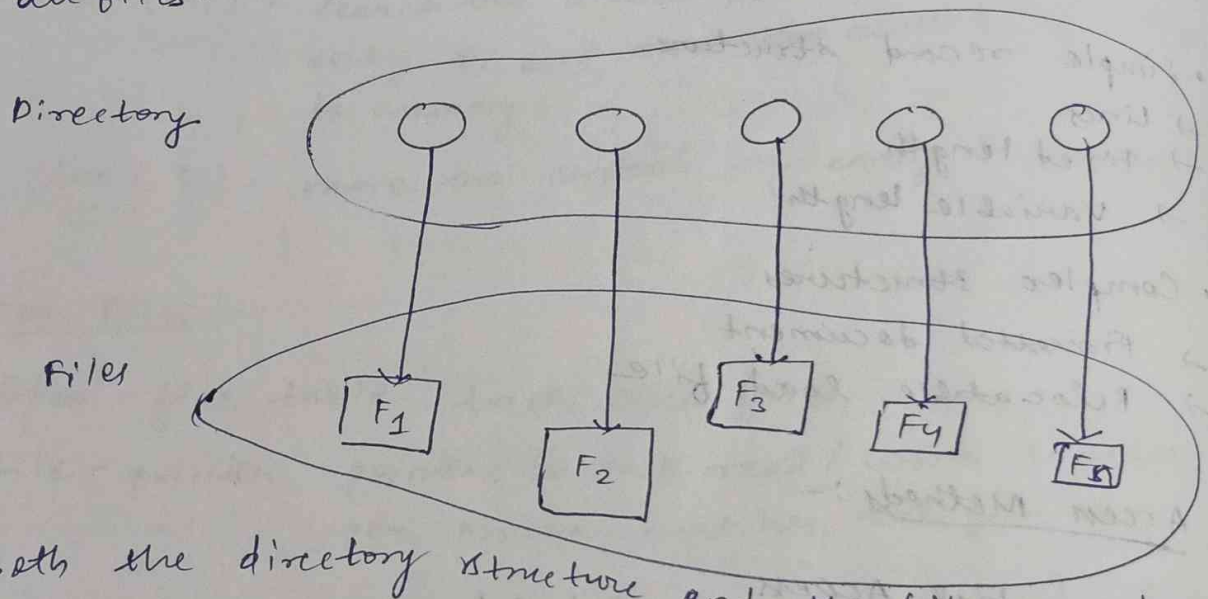
- Sequential Access
  - read next
  - write next
  - reset
  - no read after last write (rewrite)
- Direct Access - file is fixed length logical records
  - read n
  - write n
  - position to n
    - read next
    - write next
  - rewrite n

n = relative block number.

- IBM's indexed sequential - access method (ISAM)
  - → Small master index, points to disk blocks of secondary index.
  - → File kept stored on a defined key.
  - → All done by the OS.

★ Directory Structure :-

- A collection of nodes containing information about all files.



- Both the directory structure and the files reside on disk.

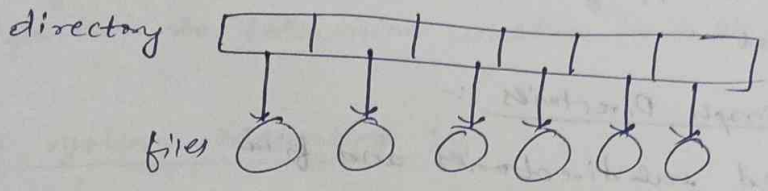
★ Operations performed on Directory :-

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system.

* Types of Directories :-

(1) Single-level Directory :-

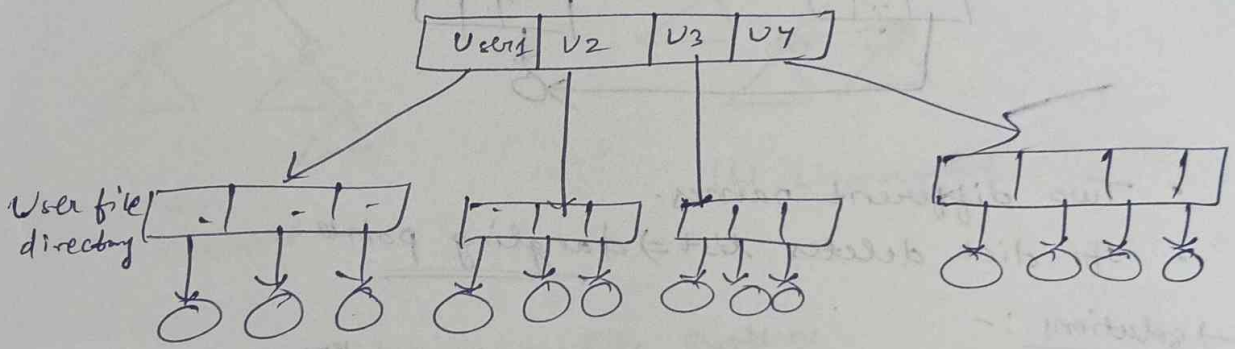
- A single directory for all users.



- Naming problem
- Grouping problem

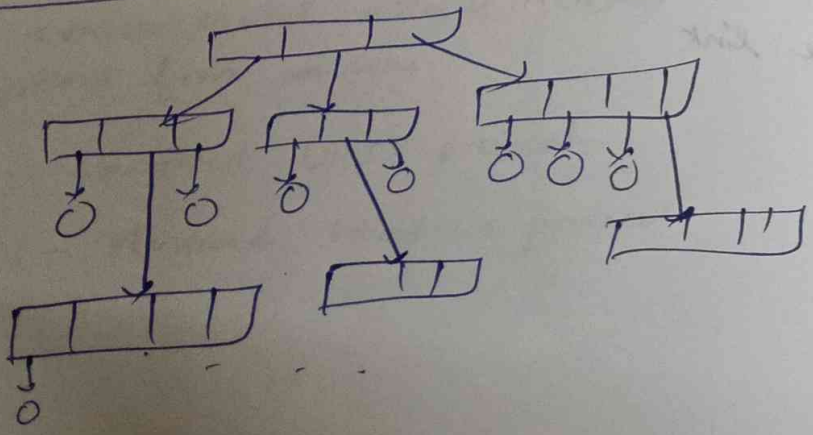
(2) Two-level directory :-

- Separate directory for each user.



- Path name
- Can have the same file name for different user
- Efficient searching.
- No grouping capability.

(3) Tree-structured Directories :



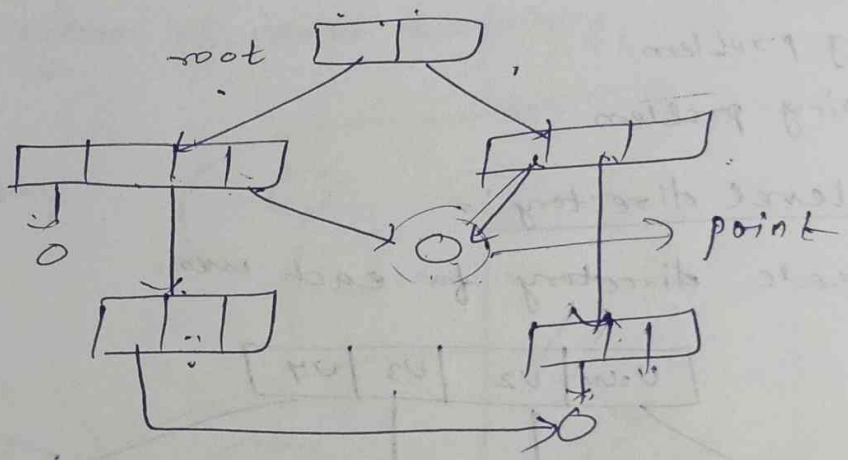
- Efficient searching
- Grouping capability
- Current directory (working directory)

→ cd / spell / mail / prog

→ type list

(4) Acyclic-Graph Directories :-

- Have shared subdirectories and files.



- Two different names.
- If dict deletes list ⇒ dangling pointer.

→ solutions :-

- Backpointers, so we can delete all pointers.
- Backpointers using a daisy chain organization
- Entry-hold-count solution.

→ New directory entry type

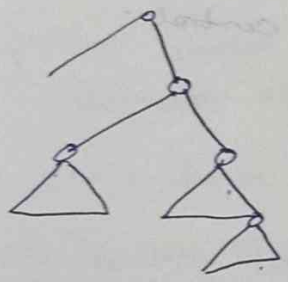
- Link
- Resolve the link

Q) How do we guarantee no cycles?

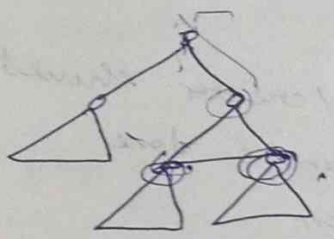
- Allow only links to file not subdirectories.
- Garbage collection
- Every time a new link is added use a cycle detection algorithm to determine whether it is OK.

* File system Mounting :-

- A file system must be mounted before it can be accessed.
- A unmounted file system is mounted at a mount point.



(a) Mounted



(b) Unmounted.

* File-sharing - Remote File Systems

- Uses networking to allow file system access between systems.

→ Manually via programs like FTP.

→ Automatically, seamlessly using distributed file systems.

- Client-server Model allows clients to mount remote file systems from servers.

→ NFS - standard UNIX protocol.

→ CIFS - standard Windows protocol.

★ File sharing - Consistency semantics :-

- Specify how multiple users are to access a shared file simultaneously.
- Andrew file system (AFS) implemented complex remote file sharing semantics
- Unix file system (UFS) implements:
  → writes to an open file visible immediately to other users of the same open file.
- AFS has session semantics.

★ Protection :-

- File owner / creator should be able to control:
  → what can be done
  → by whom
- Types of access
  → Read
  → write
  → execute
  → Append
  → Delete
  → List

File - System Implementation

~~* File-structure~~

* File - System structure :-

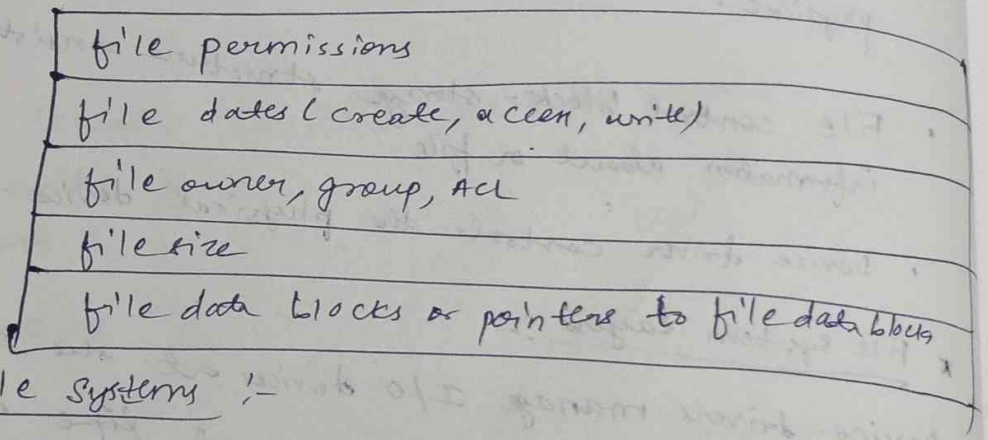
- File structure
  - → logical storage Unit
  - → Collection of related information
- File system resides in secondary storage (disks)
  - → Provided user to interface to storage, mapping logical to physical.
- File control block - storage structure consisting of information about a file.
- Device driver controls the physical device.

* File system layers :-

- Device drivers manage I/O devices at the I/O control layer.
- Basic file system given command like "retrieve block 123" translates to device driver.
- File Organization module understands files, logical addresses, and physical blocks.
  - → Translates logical block # to physical block #
  - → Manages free space, disk allocation.
- Logical file system manages metadata information.
- ~~Manag~~ Many file system, sometimes many within an operating system.

★ File-system Implementation :-

- Boot control block contains info needed by system to boot OS from that volume.
- Volume control block (superblock, master file table) contains volume details.
- Directory structure organizes the files.
- Per-file File Control Block (FCB) contains many details about the file



★ Virtual File Systems :-

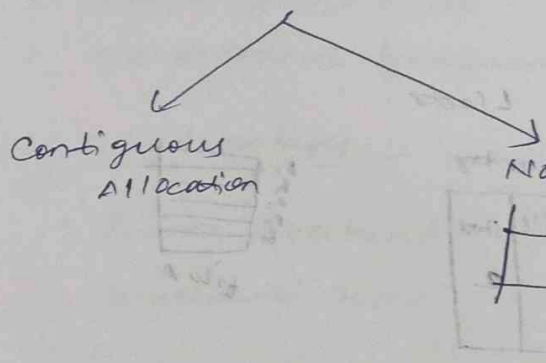
- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file system.
- VFS allows the same system call interface (the API) to be used for different types of file system.

→ Separates file-system generic operations from implementation details.

→ Implementation can be one of many file systems types, or network file system.

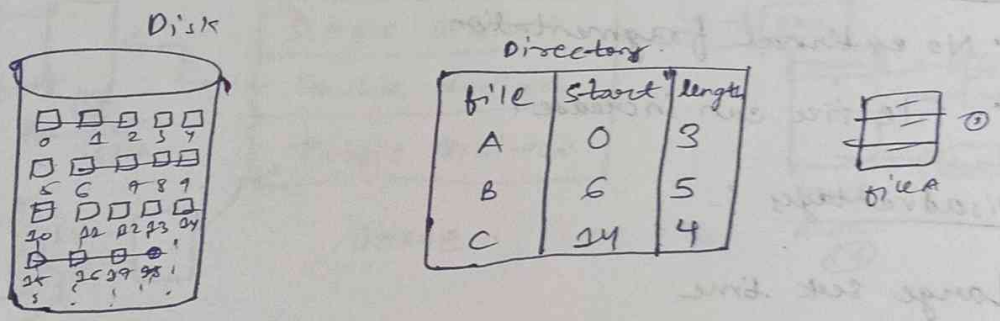
→ Then dispatches operation to appropriate file system implementation routines.

★ Allocation Methods :-



- Efficient Disk Utilization
- Access faster.

(1) Contiguous Allocation :-



- each file occupies set of contiguous blocks.
- simple - only starting location and length are required.
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line.

→ Advantages

- Easy to Implement
- Excellent Read Performance.

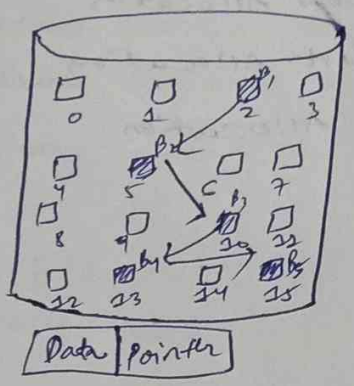
→ Disadvantages

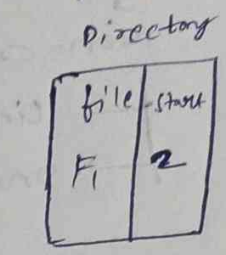
- Disk will become fragmented
- Difficult to grow file.

(2) Non-Contiguous Allocation

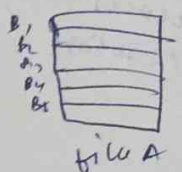
(i) Linked-List Allocation :-

each file a linked list of blocks.





Pointer



→ Advantages :-

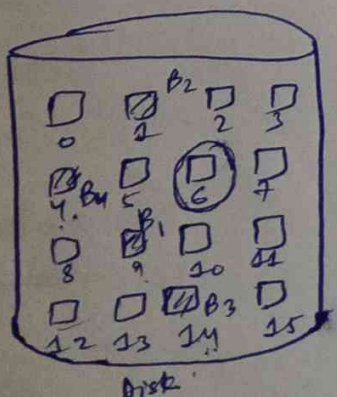
- No external fragmentation.
- File size can increase.

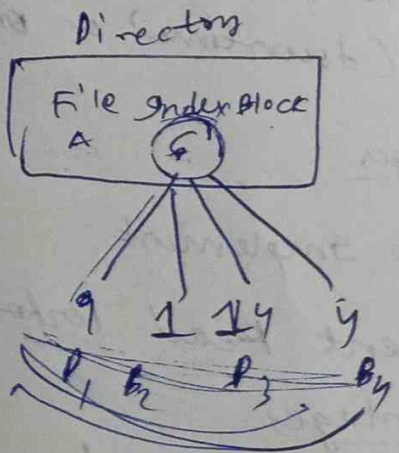
→ Disadvantages :-

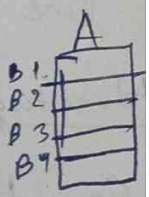
- Large seek time
- Random Access / Direct Access difficult.
- Overhead of pointers.

- FAT (File Allocation Table) variation.

(ii) Indexed Allocation :-







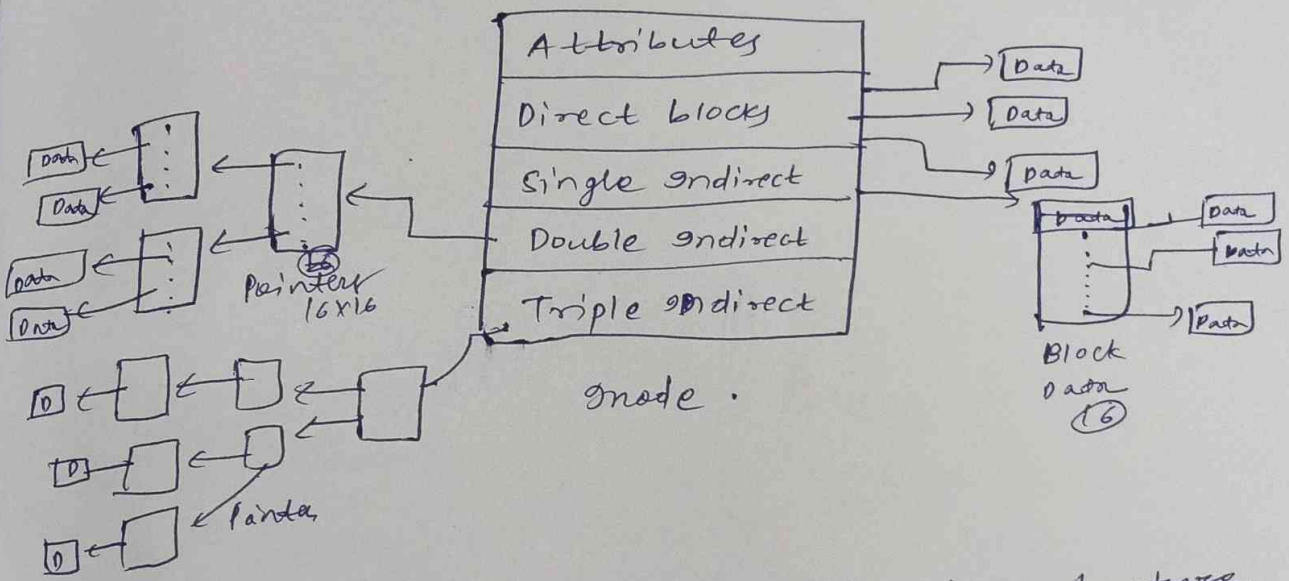
→ Advantages :-

- support direct accen.
- No external foragmentation

→ Disadvantages :-

- Pointer overhead.
- Multilevel index.

* Unix inode structure :-



Q) A file system user Unix inode data structure which contains 8 direct block addresses, one indirect blocks, one double and one triple indirect block. The size of each disk block is 128 B and size of each block address is 8B. Find the max. ponible file size?

Ans - 8 data blocks.

Size of file

$\left(8 + 16 + (16 \times 16) + (16^3)\right) \times 128B$

$= 547\ KB.$

(8 direct, 16 indi.)

Total no. of pointh in one Disk Block = $\frac{128B}{8B} = 2^4 = 16$