Disk scheduling is done by operating systems to schedule I/O requests arriving for Disk. It is also known as I/O scheduling.

SAKHEE Date: / /20 Pg.No :

Disk Access Algorithm:

→ I/O Device - Disk

→ Behavior < Read / write → Reference to a Disk.

→ Reference < Read / write

⇓

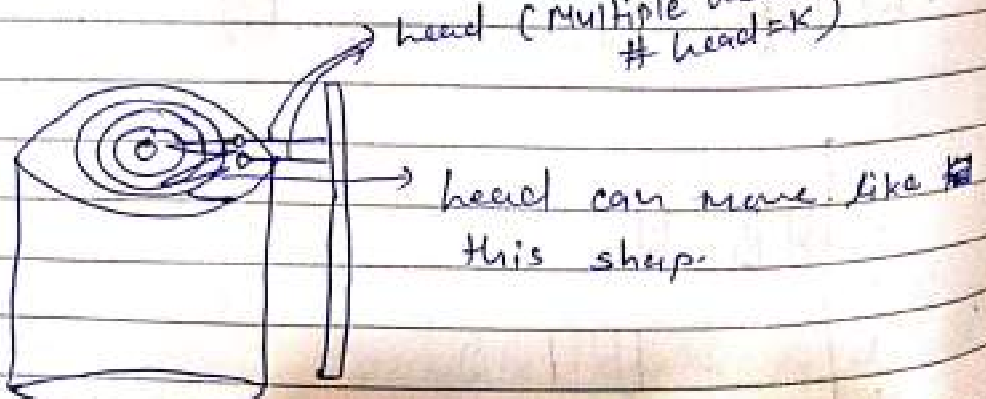
→ Series of request to the disk.
  ↳ Reference string
    ↳ Input
      ⇓
      How is the Algo. functing?

Disk:

↳ Head

→ we can have Multiple read and write in single go.
  ↳ Cylinder

⇒



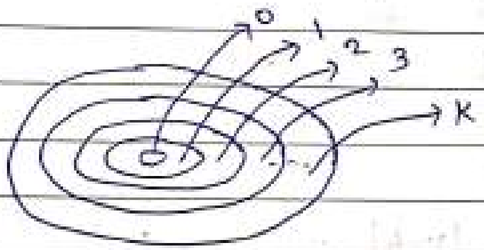
⇒ There may be multiple head to process multiple operations concurrently. But the cost we pay is much high for Multiple heads.

→ Small units : tracks, Sectors...... etc.

→ Cylinder is very big unit as it is main part of disk.

→ Tracks, sectors, ---
→ Disk blocks → each of them have size
- 512 byte
- 256 byte
- 1 KB.

→ 1 tracks → for example : 16 sectors.
→



→ Head Position:
• Disk Head : Cylinder -to- Cylinder movement = 1 unit.

Input Parameters:

→ Reference String
→ Head Position. / Initial Head position.
→ Time to move head between cylinders

* Reference String:
a, b, c, d, e, --- all are either read/write
Initial Head Position : k (unit cyl. movement = 1 unit)
↳ this process also called disk scheduling.

$K \to a \to b \to c \to d \to e$

(i) for $K \to a$

$|K-a|$ = Head movement

But we have written that it need t unit for movement of head for 0 to 1, or 1 to 2 or .... etc.

so,

Time for head movement $= (K-a) \times t$

(i) for $a \to b$

$|a-b|$ = Head movement.

⋮

$|a-b| \times t$

$|b-c| \times t$

$|c-d| \times t$

$|d-e| \times t$

O/p parameter

Total Head $= |K-a| + |a-b| + |b-c| + |c-d| + |d-e|$

$= x$

$\boxed{\text{Total time} = x \cdot t}$

O/p parameters:

(i) Total Head:

(ii) Average Head Movement $= \dfrac{\text{Total Head movement}}{\text{length of ref. string}}$

$= \dfrac{x}{5}$

☆ Example Ref String:

$\underbrace{a, a,}_{K} \underbrace{b, b, b,}_{K'} c$

0

SAKHEE
Date: / /20
Pg.No :

* (cylinder, Track)

(2,5), (3,6), (10,2)

2,3,■,10

5^{th} track of 2^{nd} cylinder.

Algorithm:

(1) FIFO : (first in first out)

↳ disk Access function : order of request.

→ for better algorith, Disk Head Movement should be less.

(2) Pick Up Algorithm. (Exa. Uber)

(3) SSF (shortest Seak first)

SSTF (shortest Seak Time first)

SSDF (shortest Seak Distance first)

(4) Elevator Algorithm. (family of Algorithm)

(5) Revised Elevator Algo.

Exa: Reference - string cylinder (0 - 199)

→ 98, 183, 37, 122, 14, 124, 65, 67.

Initial Head position : 53

FIFO:

↳ length of Ref.-Strink = 8

(All this positions are set to be available)

At t = 0, 98, 183, 37

t = 1, 122, 14, 24

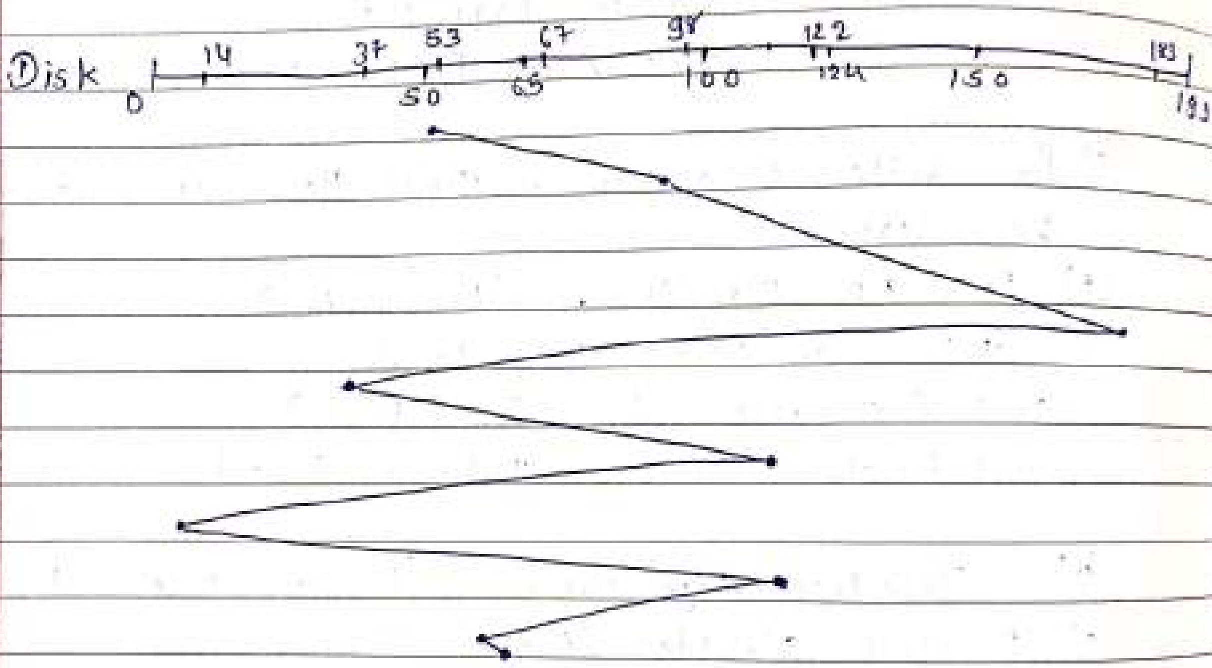
t = 2, 65, 67

not discussed here.

53 → 98 → 183 → 37 → 122 → 14 → 124 →
65 → 67.

Total cylinder Movement : = ______

☆ Disk Access chart:



⊛ Performance of Disk Access Algos:

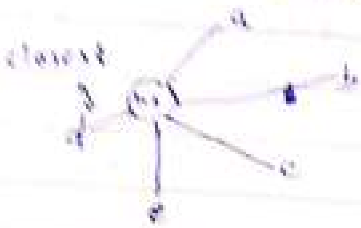
O/p {
↳ Cylinder Movement
↳ Mean / Avg. Cyl. Movement
↳ Time.

Input : Reference String
↳ Time betⁿ Cyl.-to-Cyl. Movement.

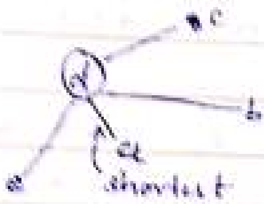
⊛ Shortest Seek First (SSF) (SSTF) (SSDF):

fun: min (Distance betⁿ two consecutive References).

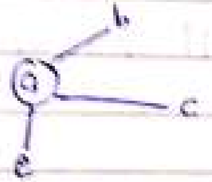
Ex: a, b, c, d, e, ...
m: initial head position.



m → d



m → d → a



m → d → a → e

o/p: m → d → a → e → b → c

③ Pick-Up:

- basis is in FIFO Algo.
- working - pickups other reference requests betn two consecutive FIFO requests.

Ex: a, b, c, d, e.
condition: c<b, b<d

→ a —— b

↳ two consecutive FIFO request.

but $c<b$, before pickuping b, system First pick-up c.

so. FIFO : a — b — c

PickUp : a — c — b → (d, e)

Exa: a, b, c, d, e.

$c<b$, $d<b$, $e<b$ | $d<c$

PickUp : a — d — c — e — b

$|(a-d)|$, $|d-c|$, $|c-e|$, $|e-b|$

⇒ Given a FIFO Reference string, the Pick-Up Algo. picks-Up all request betn two consecutive FIFO requests.

Exa: 98, 183, 37, 122, 14, 124, 65, 67.

Head position - 53

O/p: 53 —— 65 —— 67 —— 98 —— 122 —— 124 —— 183 —— 37 — 14

Total Head Movement:

$= |53-65| + |65-67| + |67-98| + |98-122| + |122-124| + |124-183| + |183-37| + |37-14|$

⇒ In case no pick-Ups could be ~~continued~~ carried out for a given reference string, Pick-up Algo. reduces to a basis FIFO Algo.

⊛ Elevator Algorithm:

↳ ① SCAN Algo.

① SCAN Algo. }
↳ ② LOOK Algo. } direction of head movement.

Disk

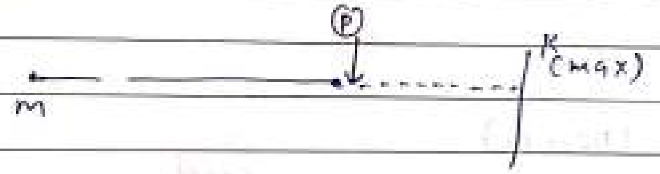
X ← DN (down) — $\boxed{M}$ — UP → K

X < M   K > M

→ SCAN Algo: (default) SCAN (UP bit) ↳ Look (UP)
SCAN (Down bit) ↳ Look (DOWN)

Initial Direction of Movement.

☆ SCAN (UP):



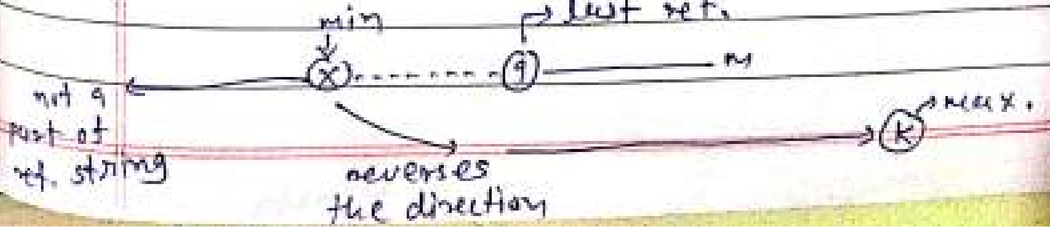
Ⓟ :→ last req. that could be fulfilled in UP direction. But system actually check to the last point at K.

→ K is not a part of reference string. K is last cylinder in Up direction.

Ref: M - - - - - P ⟶ Ⓚ Max

X ⟵ reverses the direction (DOWN)

☆ SCAN (DOWN):
↳ Reverse of the SCAN (UP):



SAKHEE
Date: / /20
Pg.No :

* → SCAN (UP) = 1
↳ set in UP dire.

SCAN (DOWN) = 1
↳ set in Down dire.

→ overhead = $\left.\begin{matrix} |p-k| \\ +|q-x| \end{matrix}\right\}$

Exa:
98, 183, 37, 122, 14, 124, 65, 67
SCAN (UP),
Cylinder : (0 to 199), H: 53

(UP=1)
53 → 65 → 67 → 98 → 122 → 124 → 183
overhead
14 ← 37 ← 199
(max)
(DN=1)

SCAN (DOWN) :
H : 53
overhead
53 (DN=1) → 37 → 14 → 0 (min) → (UP=1) 65 → 67
183 ← 124 ← 122 ← 98
(don't go to 199)

End-Sem

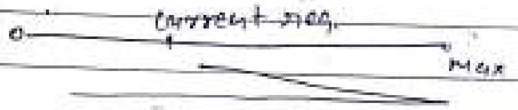
⊛ Elevator Algo: (family of Algorithms)

- SCAN (UP/DN)
- Look (UP/DN)
- C - SCAN
- C - Look

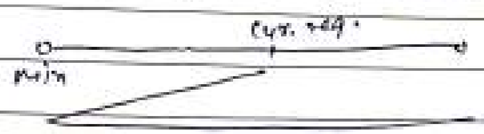
→ Avg. Disk movement will decrease (not always the case)

☆ SCAN :

- UP



- DOWN

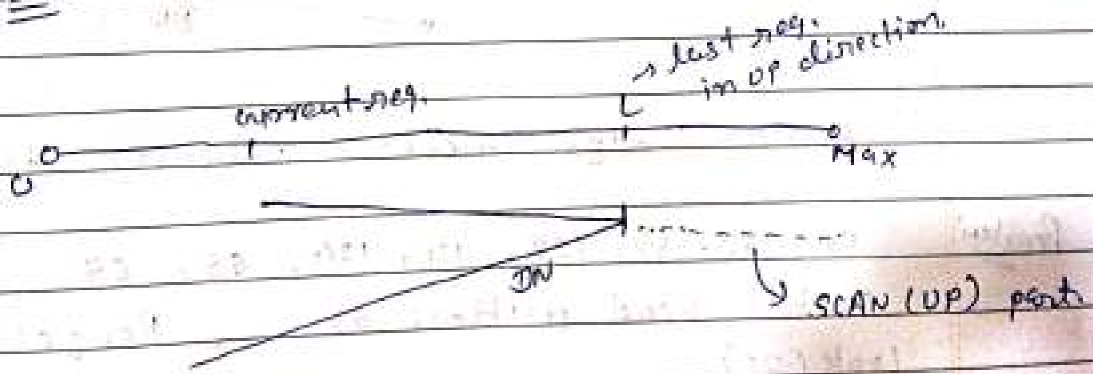


⊛ overhead (increase Disk movement) .

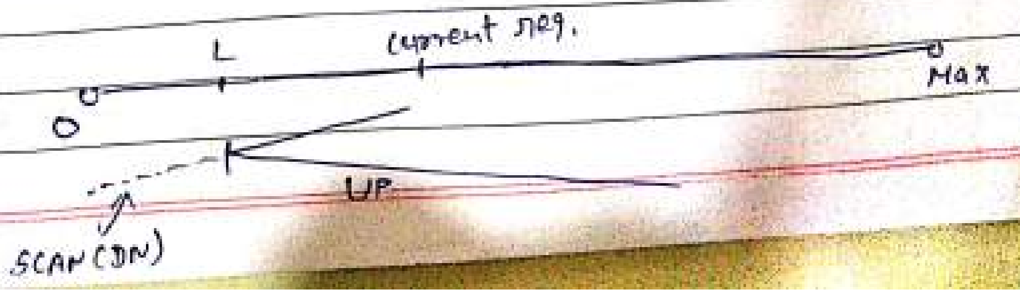
⊛ LOOK:

→ Improved SCAN Algo.
→ Go to Max or 0 cylinder / location on disk.
→ instead of MAX or 0, it will only read to the last request in either of the direction.

Look (UP):

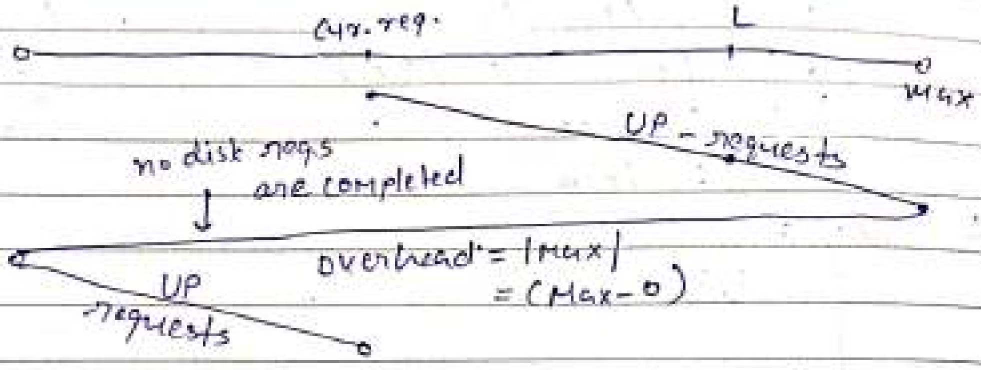


Look (DN):



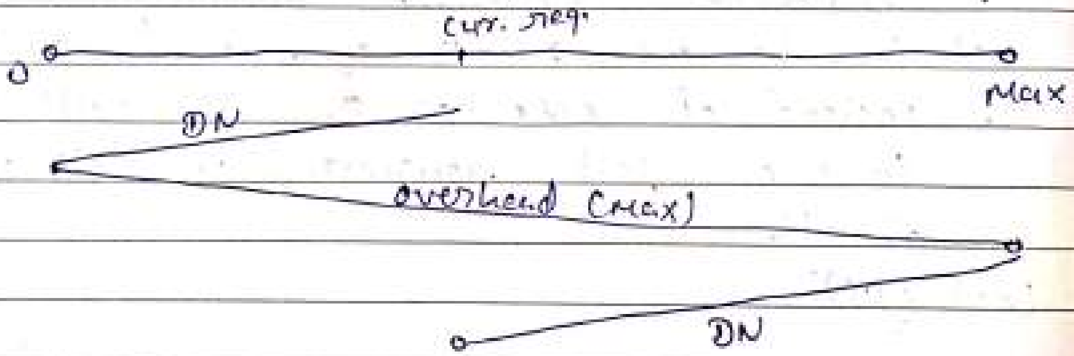
because it looks only in one direction.

⊛ C-SCAN: ✗ [C.SCAN(UP), C.SCAN(DN)]   ✓ [SCAN(UP), SCAN(DN)]



⇒ Considering max and 0 as adjecent then overhead equals to one. so this is improved version. overhead = 1

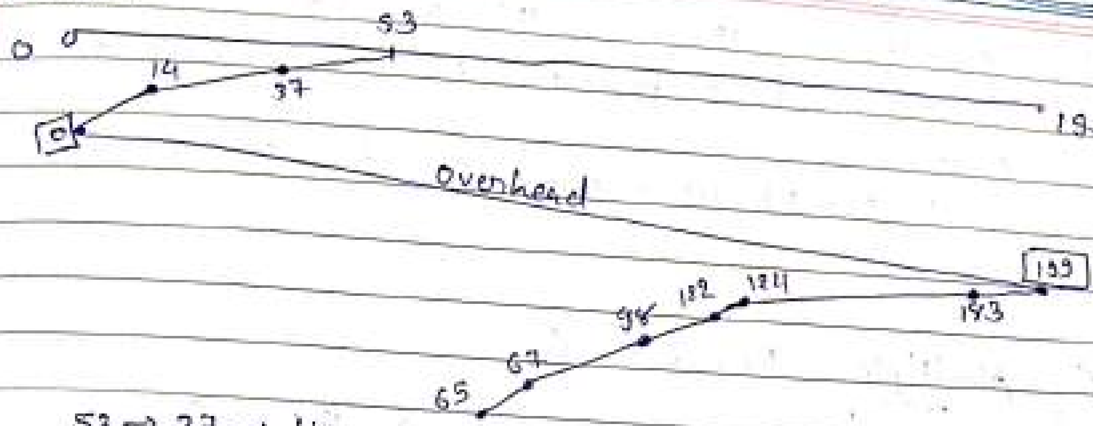
⇒



→ C-SCAN algo. never reverses its direction.

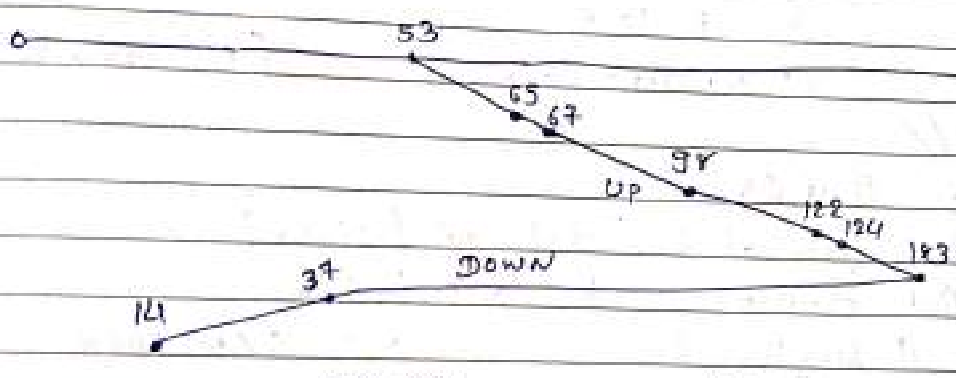
Problem: 98, 183, 37, 122, 14, 124, 65, 67
Initial head position: 53 , Range: 0-199
Look (UP)
C-SCAN : Initially moving in down direction.

→ C-SCAN:



53 → 37 → 14 → $\underbrace{\to 0 \to 199}_{\text{overhead}}$ → 183 → 124 → 122 → 98 → 67 → 65

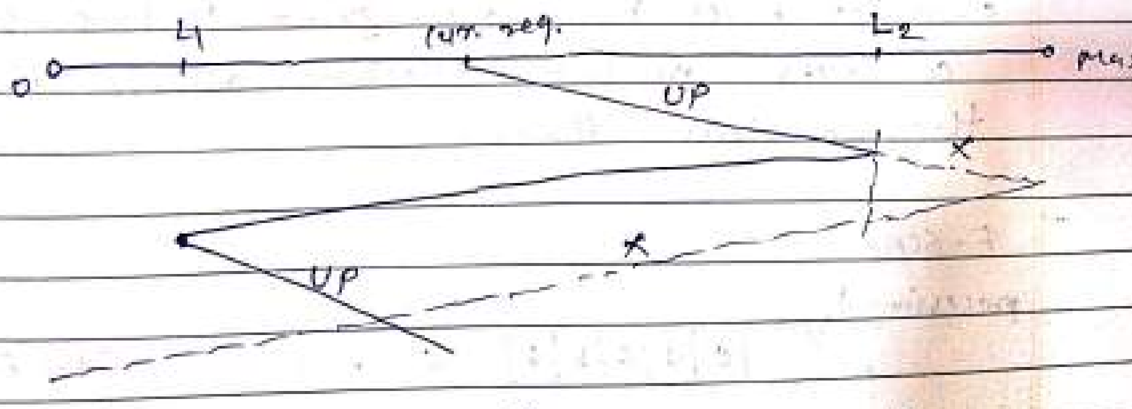
Look (UP):

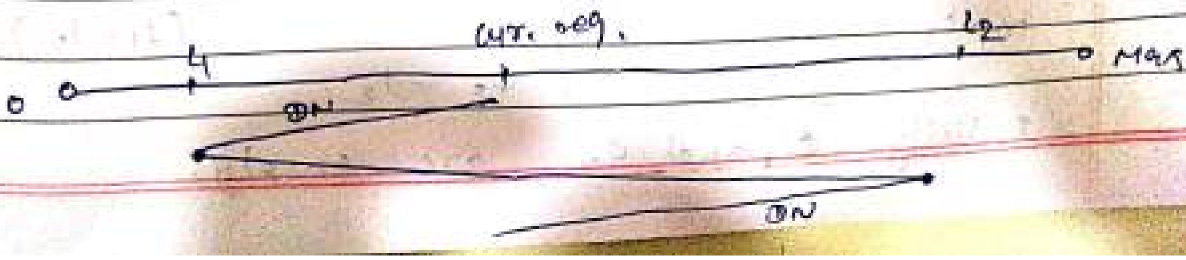


$\frac{199-37}{\text{SCAN}}$ (Max diff) $\frac{183-37}{\text{Look(UP)}}$ (Max diff)

→ 53 → 65 → 67 → 98 → 122 → 124 → 183 → 37 → 14.

⊛ C-Look:





Prob: 98, 183, 37, 122, 14, 124, 65, 67
initial head position : 53 , Range: 0-199
C-look.
↳ 53 - 37 - 14 - 183 - 124 - 122 - 98 - 67 - 65

⊛ Multiple Request Q.

↳ F-SCAN → by default: 2 Qs.
↳ N-step - SCAN.
↳ F- look
↳ N-step- Look (N: number of Qs)

N → # of Qs
→ length of Q (defined / given)
Ref. string: a, b, c, d, e, f, g, h
Q length = 5 ↳ SCAN / Look
$\underbrace{a, b, c, d, e}$, $\underbrace{f, g, h}$
5 req. 3 req. (remaining)
SCAN / LOOK SCAN / LOOK
↳ initial head position / direction.

→ In case of, Multiple Qs, whenever the disk scheduler moves from one group to the other, it continues in the direction where it left in the previous Queue.

⇒ F-SCAN

processing Q → | e | d | c | b | a | $Q_1 = l_1$ (for Exg: $l_1 = 5$)
input → $Q_2 = l_2$ $(l_1 = l_2)$

Initially Q_1, and Q_2 are empty.

→ Q_1 : initially input Q.
→ post Q_1 (filled) → processing Q
& Q_2 → input Q

→ then, later, Q_2 (filled) → processing Q
& Q_1 → input Q.

(*) N-step-SCAN:
↳ Drawback : F-SCAN

Ref. string: a, b, c, d, e, f, g, h, i
Q: length = 4

d	c	b	a	Q_1
h	g	f	e	Q_2

direction

(i) → waiting Queue

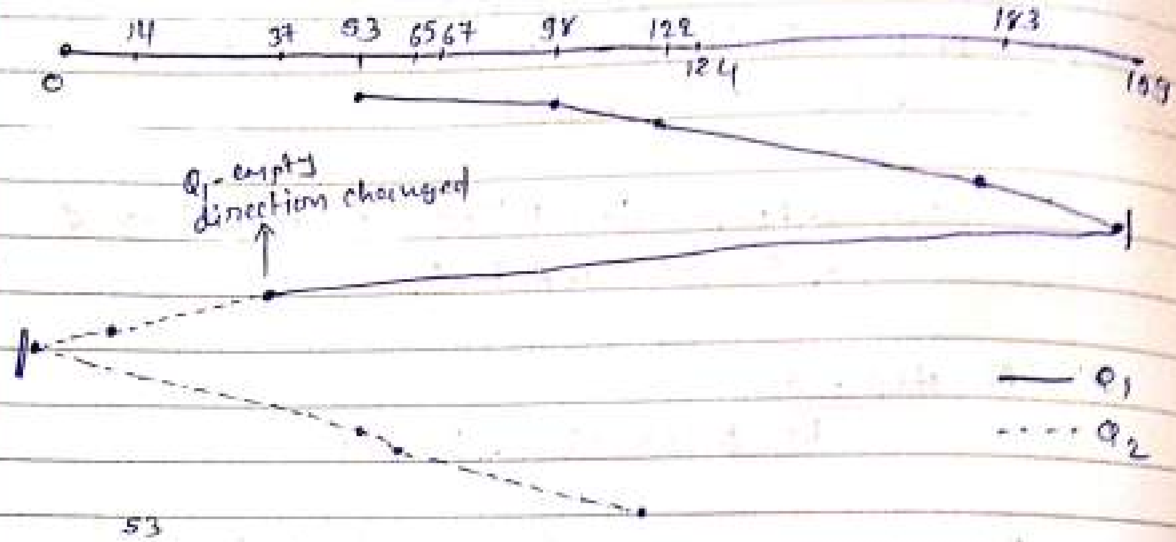
if we have N-Queues

	i	Q_3

Exa: 98, 183, 37, 122, 14, 124, 65, 67
Initial head position: 53 Range: 0-199
F-SCAN (Q length = 4)
3-step-SCAN (")

(1) → Moving in ~~down~~ up direction

122	37	183	98	Q_1
67	65	124	14	Q_2



53
↓
F-SCAN : 98 , 122 , 183, 199, 37, 14, 0, 65, 67, 124

* Linux - have multiple disk scheduling policies.
  - policy : 1 (current)
  - Policy : 2
  - policy : 3

Disk Request → change the policy using command line
(CFQ)
(Completely Fair Queuing)
↓
Following Disk request

* Benchmarks:

  ↳ 1-GB - video file
  ~~[illegible]~~ want move this file from pc to Pendrive
  policy 1 policy 2

→ once the device sends data back to the driver, the driver
may invoke routines in the original calling program.
Drivers are hardware dependent and operating-system specific.
They usually provide the interrupt handling required for any
necessary asynchronous time-dependent hardware interface.

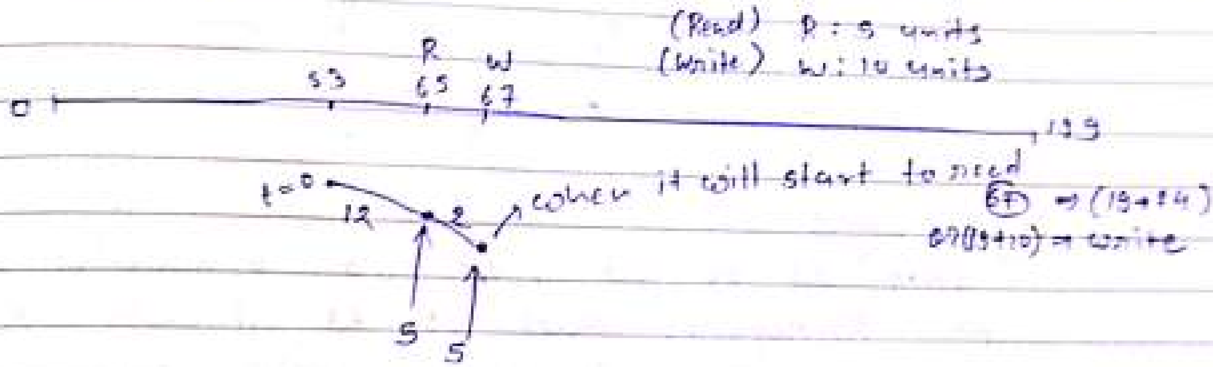
SAKHEE

⊛ Deadline Driven

↳ C-SCAN (Default)

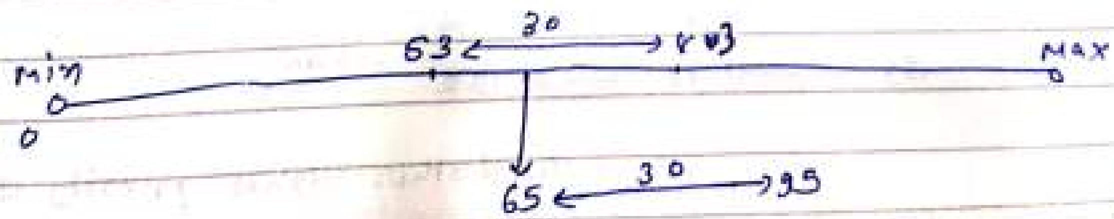
↳ If there is a risk of a disk request
a given deadline, it priorities that Queue.

↳ Necessary to check that while C-SCAN is running. new disk request cross the given deadline.



⊛ Anticipatory Scheduling:

→ wait for certain period for a given locality range
Else,
C-SCAN / C-LOOK (given)



→ waiting for some time if any request betn x to x+30 is come up.
53 → 69 and 65 → 67

if ① fails
↳ Default ~~CSAN~~ (CSCAN / CLOOK)

→ Locality Range is moving w.r.t current ~~current~~ head position.

⊛ CFQ. (Completely Fair Queuing)

98, 183, 37 (RT), 122 (RT), 14, 124, 65, 67

→ from ^moving 53 to 63, at 63 scheduler check for RT request and at 63 already first 3 requests arrived. so it will take 37 (RT) first.

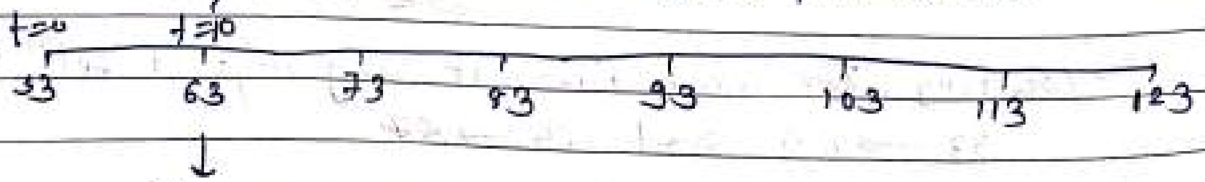
AT: 0, 4, 9 (RT), 12 (RT), 15, 18, 20, 20

→ A filter with 10ms value ^of time period. will check a real time cost of requests at every 10 ms.

53 ⟶ 98 X

53 ⟶ 37 ⟶ 122 $\xrightarrow{\text{C-SCAN}}$

RT ⇒ check other higher priority Qs.



from 63 cylinder head will move to 37.

* A deadlock is a situation in which two programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

SAKHEE
Date:
Pg.No :

Deadlock Handling : (out of problem)

↳ Process → No progress.

↓

system Resources → Problem

sharable
All K processes are using the resources.

non-sharable
(one process at a time)
↳ (K-1) (wait for Resources)
↓
Infinitely
↓
Deadlock.

* Methods for Deadlock Handling :

(1) Ignore DLs.
↳ Unix/Linux based system ~~actully~~ actually do this.
↳ possibilities of DLs is very less.

(2) Deadlock Prevention.

(3) Deadlock Detection & Recovery.

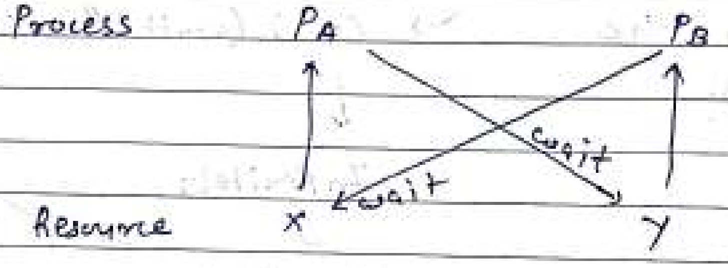
(4) Deadlock Avoidance.

* Deadlock - Condition :

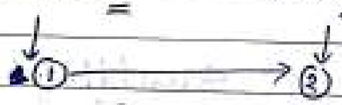
SAKHEE
Date: / /20
Pg.No :

(1) Mutual exclusive
(2) Hold and wait → Hold and wait for resources until process get it
(3) No Preemption.
↳ resource gets block until process releases resources itself.
(Not releasing held resources)

(4) Circular wait



⊛ Detection & Recovery:



↳ Detection is done by use of Graph.
→ Graph of process & Resources
↳ Resource Graph
or Resource allocation Graph (RAG)

⊛ Methodology:

Maintaining RAG (1) Construct RAG Q.→ How?
↳ it is not one time scenario. it is continuously updated. as usage of Resources changes.

Putting all Resources Separate in RAG.

Detection. (2) At any time instance.
↳ check, RAG has a cycle.

Reco

Recovering (3) Recovery
- How to break the cycle?
- Has two methods For Recovery.

⊛ RAG (Resource Allocation Graph):
- Two elements → nodes, edges.

⇒ what type of nodes and edges ~~RAG has?~~ are in RAG?

(1) Types of nodes: Two types

- Process nodes → always independent
  - (Pi)
- Resource nodes → may be dependent.
  - $\boxed{Ri}$
  - # Resource nodes decide type of nodes.

→ ways to Represent Resource nodes

e.g.

10 printers:

has lot many nodes → $\boxed{R_1}$ - - - $\boxed{R_{10}}$

OR $\boxed{\cdots}$ R → Preferred one

Putting all Resources separate in RAG.

⇒ when we doing RAG manually then $\boxed{R_1}$ - - $\boxed{R_m}$ is preferred method.

otherwise. $\boxed{\cdots}$ is preferred method.

- For Automated RAG.

Scanned by CamScanner

(2) Types of edges:

↳ connection – Directional
↳ betn process & Resource node

Pi is requesting Resource Ri

$(P_i) \longrightarrow \boxed{R_i}$ Request edge

$(P_i) \longleftarrow \boxed{R_j}$ Assignment edge
(Allocation edge)

(R_j is assignment to P_i)

Two types : ① Request edge
② Assignment edge.

⇒ Cycle : Across all P_i & R_i

updating of RAG.
- ↳ Process node get deleted on process termination.
- ↳ Resource node & process node assignment edges get deleted after resource usage.
- ↳ If a resource is removed from the system the Corresponding Resource node is also removed from RAG.

Recovery:

↳ Process based
↳ Resource based } two types

Process based:
⇒ On deletion of a cycle in RAG.
↳ lets we have two process in cycle (deadlock)
(P_T, P_j, P_K)

↳ we need to kill one process. but it does not give surity that cycle will get break after killing one process.

→ Two ways of killing
(1) Kill all process in cycle
(not good idea, because it will kill some useful processes)

(2) Incremental kill
↳ need to use a criteria to select a process to kill
↳ it is design parameter based AT, ST, etc.
requested

Resource based:

(1) Remove Request edges
↳ it will not create problem because process can request again in later stage

(2) Remove assignment edge
↳ incremental

→ In case of assignment edge
↳ A process that has a resource assigned, but has not started using the resource. the resource removal of that assignment edge is preferred.

Incremental (Preferred) = Insuring that a process has minimal loss of computation and Resource usage

⇒ Banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

⊛ ⇒ Deadlock Avoidance:

↳ Good algorithm. (optimal approach), futuristic

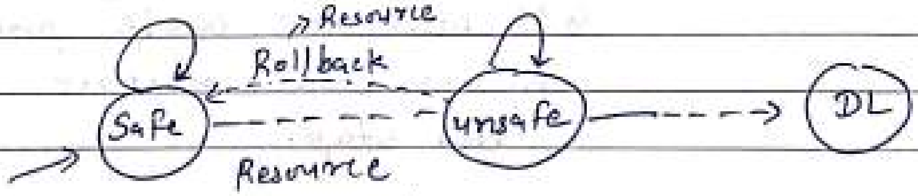
↳ Implementation is difficult

→ it assumes that the OS is fully aware of all resources that a process will use in its life cycle.
↳ if this is achievable then implementable

Algos: ⇒ System : Matrix
↳ Detect safety.
↳ safe / unsafe

safe ↓ No deadlock ↳ can avoid deadlock

unsafe → may has deadlock (Any incorrect Resource assignment)

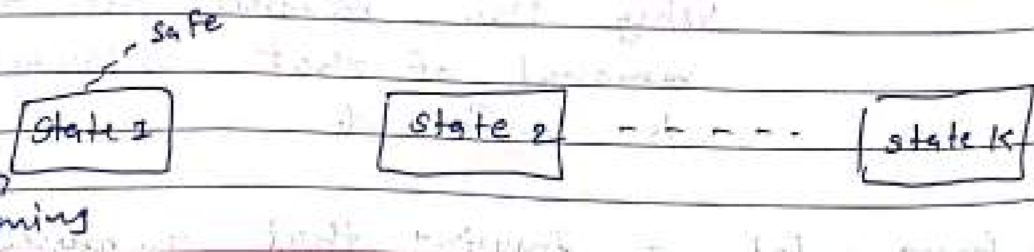


Resource Assignment issues ~~inprocesses~~

Important output of deadlock avoidance

(Banker's Algorithm)

⊛ ⇒ Resource Assignment Problem:



Resources
→ Free Resources

safe

state : 1 free

P_1 — $R_1 R_2$

P_2 — $R_3 R_4$ → free ⟹

lets P_1 does not Require R_1

P_2 does not Require R_4

safe

state : 2

P_1 — R_2

P_2 — R_3

→ state : 3

when new process P3 created

lets state k is unsafe

Roll back

$\boxed{\text{state : k}}$

(3) Maintains different Matrices:

(1) ↳ Resource (Related to Resources)

$$\begin{array}{ccccc} R_1 & R_2 & R_3 & --- & R_K \\ \end{array}$$

$$[\; x \quad y \quad z \quad ---- \quad m \;]$$

⇓

$$[\; 1 \quad 2 \quad 4 \quad --- \quad 1$$

↳ two copies of Resource R_2

⇒ This is $(1 \times K)$ matrix where K is types of Resources.

(2) Assignment of Resources:

$$\begin{array}{c|cccc|} & R_1 & R_2 & \cdots & R_k \\ \hline P_1 & & & & \\ P_2 & & K & & \\ P_3 & & & & \\ \vdots & & & & \\ P_z & & & & m \end{array}$$

→ Process P_2 is using K copies of ~~Resour~~ Resource R_2

→ P_z is using m copies of Resource R_k

⇒ Algo / system has to ~~beware~~ be aware of maximum resource requirements for all processes.

$$\begin{array}{c|cccc|} & R_1 & R_2 & \cdots & R_k \\ \hline P_1 & & & & \\ P_2 & & n & & \\ \vdots & & & & \\ P_z & & & & \end{array}_{z \times k}$$

→ P_2 need atmost n copies of Resource R_2.

⇒ Every value is the Max. Resource Requirement matrix is the max. resource req. for every process.

⊛ Need Matrix

↳ Represents the resources still needed.
↳ Diff : Max. Resources − Assigned Resources
$z \times k$ $z \times k$

Need:

$$
\begin{array}{c|c}
 & R_1 \quad R_2 \quad \cdots \quad R_k \\
\begin{array}{c} P_1 \\ P_2 \\ \vdots \\ P_z \end{array} &
\left[\begin{array}{cccc}
 & & & \\
0 & 0 & \cdots & 0 \\
 & & & \\
 & & &
\end{array}\right]_{z \times k}
\end{array}
$$

→ P_2 has received all needed resources
→ P_2 can finish and free the assigned resources.

⊛ Available Resources:

↳ Resources which are currently free.

Assignment Resources

$$
\begin{array}{c|c}
 & R_1 \; R_1 \; \cdots \; R_k \\
\begin{array}{c} P_1 \\ P_2 \\ \vdots \\ P_z \end{array} &
\left[\begin{array}{c}
1 \\ 1 \\ 1 \\ 1
\end{array}\right]
\end{array}
$$

→ Sum up the columns. R_1 resource used by P_1 to P_z, sum it and substract it from system Resources. it gives currently free copies of particular Res.

→ For every row in Need Matrix, compare it with currently available resource matrix.

↳ atleast one row in need Matrix should be less than or equal to currently available Matrix.

$\boxed{\text{state: 1}} \xrightarrow{P_0 \text{ termination}} \boxed{\text{state: 2}} \xrightarrow[\text{to } P_1]{\text{Assign Resources}} \boxed{\text{state: 3}}$

[1 5 2 0]

$$\begin{array}{r} 1\,5\,2\,0 \\ 0\,0\,1\,2 \\ \hline 1\,5\,3\,2 \end{array}$$

[1 5 3 2]

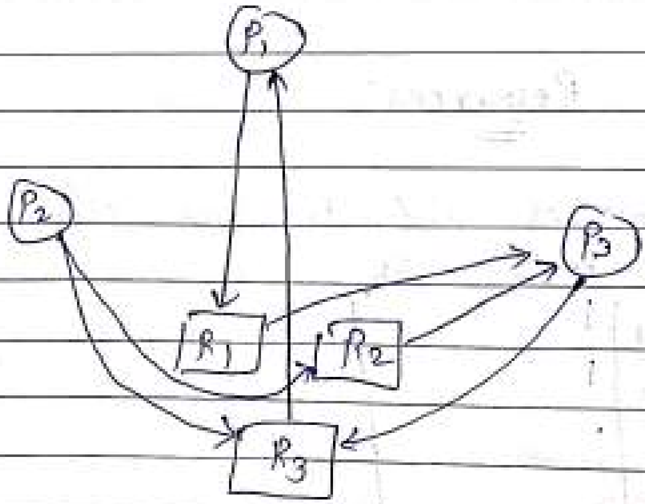
$$\begin{array}{r} 1\,5\,3\,2 \\ -\,0\,5\,1\,0 \\ \hline 1\,0\,2\,2 \end{array}$$

Exs:

$P_1 \xrightarrow{\text{Req.}} R_1$ Assign R_3

$P_2 \xrightarrow{\text{Req.}} R_2, R_3$

$P_3 \xrightarrow{\text{Req.}} R_3 \xrightarrow{\text{Assig.}} R_1\ R_2$



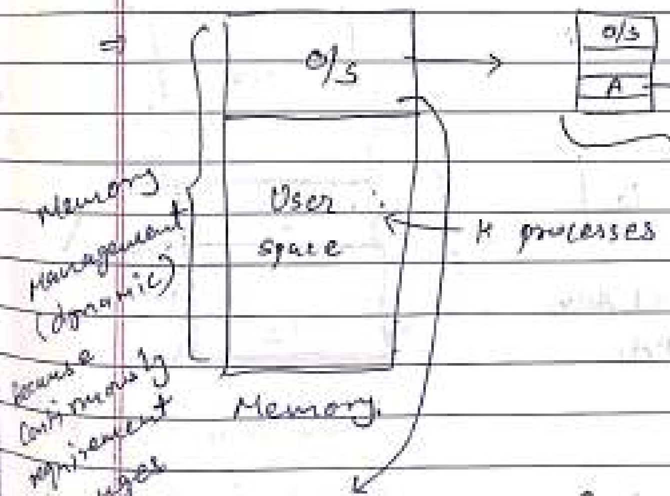
⊛ Memory: ↳ Primary Memory

CPU $\xrightarrow[\text{slow}]{\text{faster}}$ Disk : Secondary Memory

Hierarchy of Memory:

→ Primary Memory : Faster in access
↳ Volatile
↳ How OS manage this resources?
↳ Understand Memory Management Techniques.

why?
↳ Memory is limited.
↳ Need to use resource increases.
↳ Complexity of application.
↳ Multiple processes using memory (shared resources)



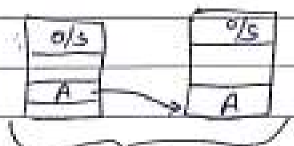


Fig: ①

Memory Manager (what is set of duties perform by this Memory Manager)

Duties
(1) Allocation of Memory
↳ Following certain rules
(2) De-Allocation (free-up) of Memory
(3) Reallocation (depend on availability of Memory and addressing may be diff.)
(4) Sharing of Memory space
(5) Relocation (Fig: ①)
↳ Relocates Process one place to another place.

(6) Protection & ~~Primary~~ Privacy

⊛ Terminology:

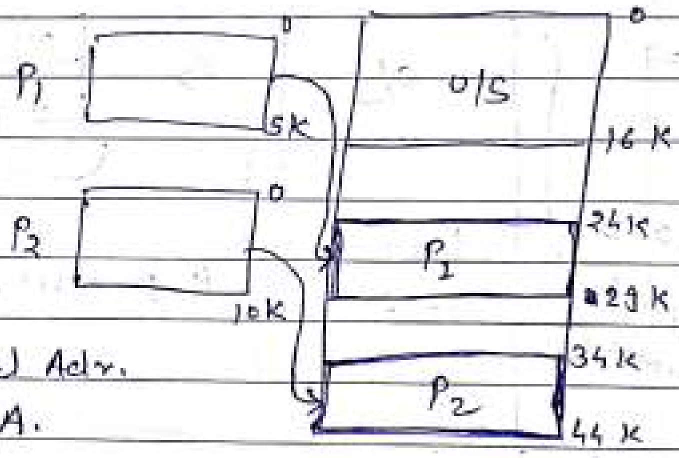
(1) Address Translation:

Programmer / User → logical Address space (LA) (all c, c++, java programs)

System → Physical Address (PA) space.

Translation: LA → PA

starts: 0 ⟶ size : (end)



(0-5K, 0-10K) : logical Adr.
(24K-29K, 34K-44K) : PA.

↳ we can use PA which is beyond 16K.

(2) Unit of Addresing :

Memory ⟶ Block
(chunk of memory)
(group of memory location)

→ Block is primarily used in context of Disk.

→ Complex system may contain multiple type of pages / frame (2K, 4K, 8K, ...).

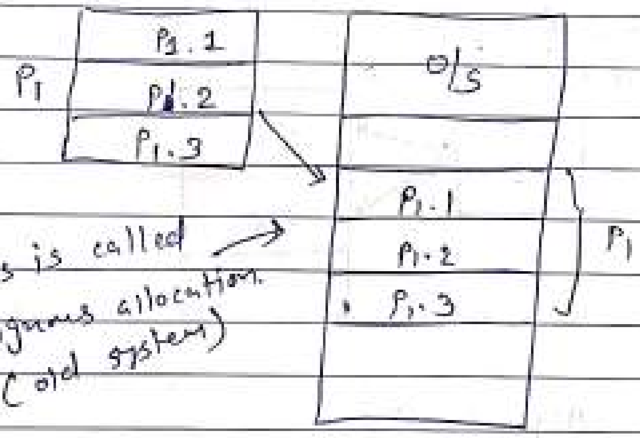
~~Memory~~ Memory → Partition
chunk of Memory → Read / write

→ Units : → Page / frame or page frame
→ Segment

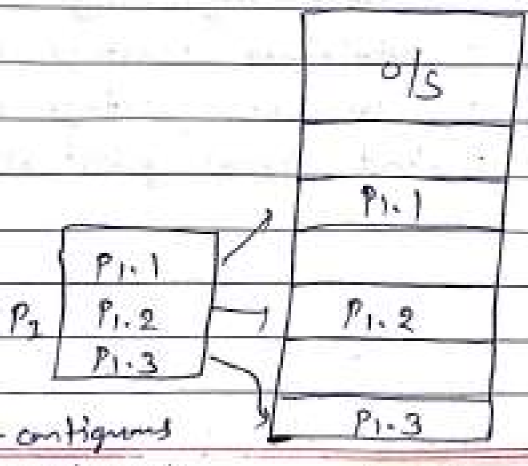
→ depending of type of unit there are different Memory Management schemes.

⊛ Classification:

Memory Management Techniques

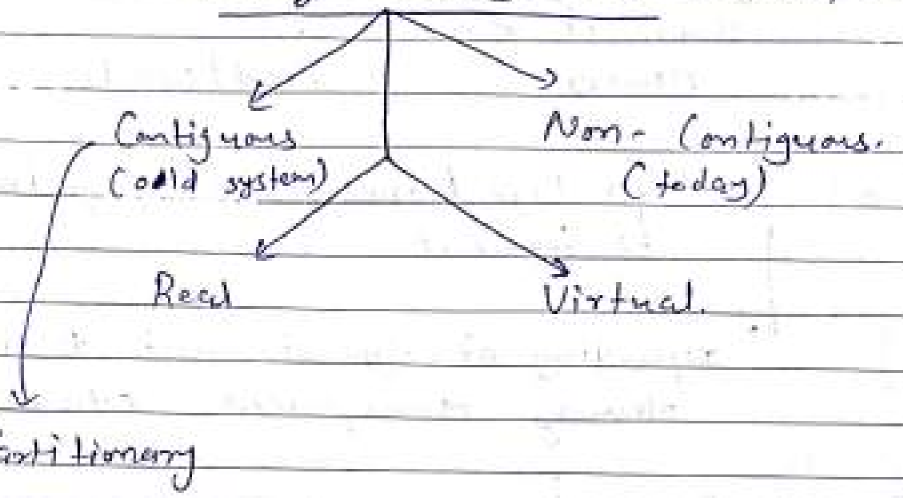


This is called contiguous allocation. (old system)



non-contiguous allocation. (new systems)

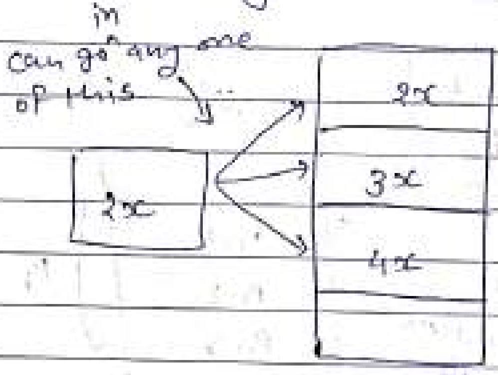
Memory Management Techniques



Date: 22/4/19

Problem:

↳ Program can go into multiple partitions.



⊛ Placement Algorithm:

sub optimal

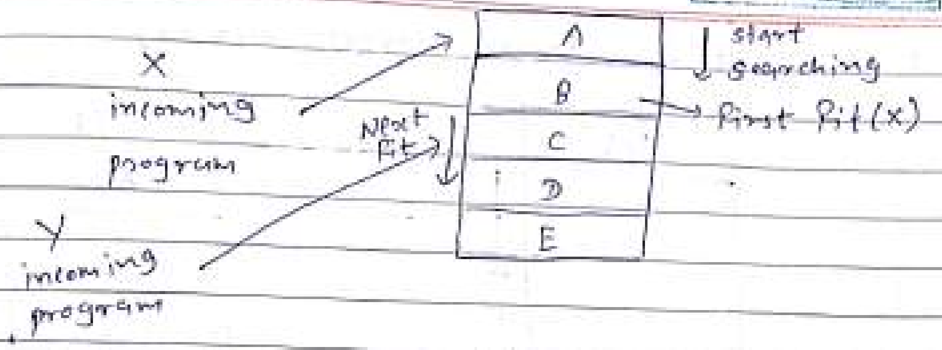
① First - fit : from beginning

Best possible fit → (2) Best - fit : Minimum wastage of Memory

(3) Worst - fit : Maximum wastage of Memory.

(4) Next - fit : start from point of last allocation

(1) First- fit



→ here X using A and B, ~~then~~ if Y is following Next-Fit, then it start searching from last allocation means from C.

Z incoming program → A-Z, B-Z, C-Z, D-Z, E-Z

A-Z, B-Z → already allocated

C-Z → BF. then Z 4 wait for C.

if Best is D that means BF is available then Z go into D.

sub optimal → { Best Possible, Best Available } → different then Best Fit (Optimal)

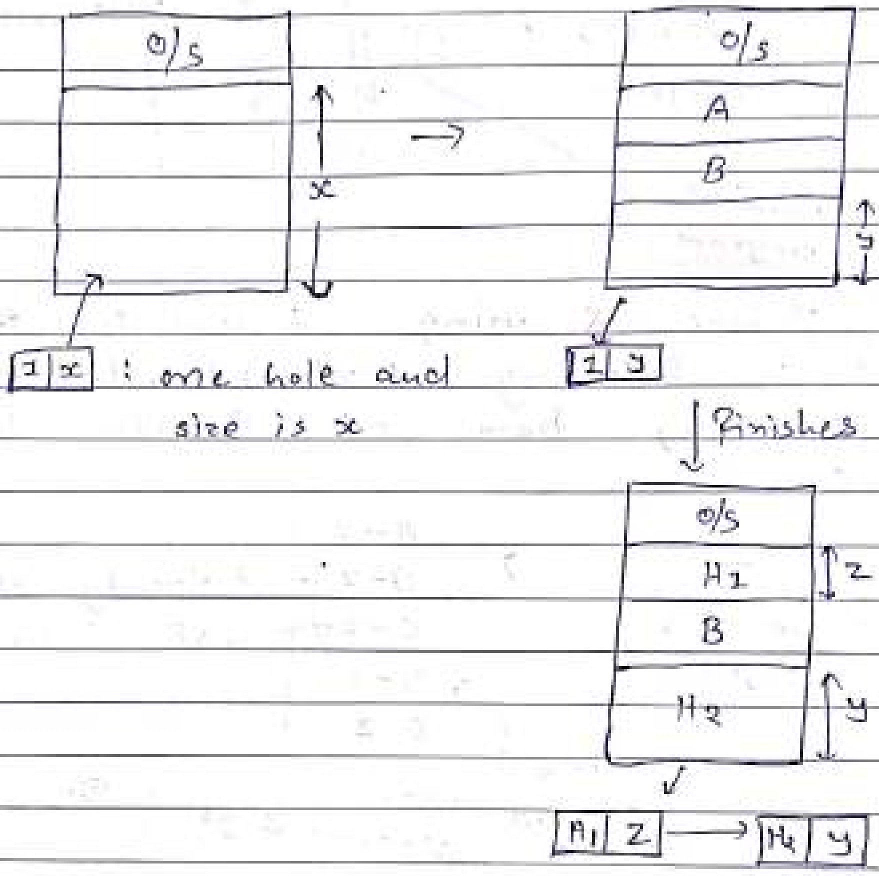
Worst-Fit : largest space

⊛ Variable Partitioning:

↳ Hole (free space) → every memory allocation creating a partition & a hole.

$$H \xrightarrow[A]{\text{allocate}} \left.\begin{matrix} A \\ H-A \end{matrix}\right\}$$

→ Hole list : scattered across memory.

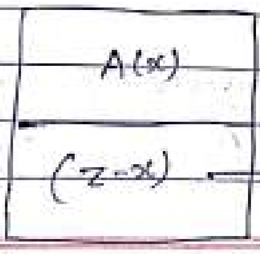


→ Placement Algo :→ Fixed Partitioning (Partition)
→ Variable Partitioning (Hole)

⊛ Memory Fragmentation: (Wastage of Memory)

① Internal Fragmentation (IF)
(Inside a partition/page - i.e. Memory block)

A
(size = x)
(z > x)



— IF (waste of Memory)
~~(Fixed partitioning)~~

Partition size = Z

larger hole z

A(x)
z-x

z-x → becomes a new hole in variable partition

⇒ All memory partition are created to the size of powers of 2 (closest)

A (20K), lets say 3 bytes, but its ideally assigned 4 bytes

$\boxed{3 \text{ byte}} \longrightarrow \boxed{4 \text{ byte}}$ → (1 byte of IF)

$\boxed{5 \text{ byte}} \longrightarrow \boxed{8 \text{ byte}}$ → (3 byte of IF)

Theoretical: No
Practical: Yes

So, for "Variable partitioning there is no Internal fragmentation" statement is false.

Conclusion: Both Fixed & variable partitioning have IF

(2) External Fragmentation: (outside the partition)

O/S	
H_1	$H_1 < D$
A	
H_2	$H_2 < D$
B	
H_3	$H_3 < D$
C	

D incoming $\xrightarrow[\times]{\text{load}}$

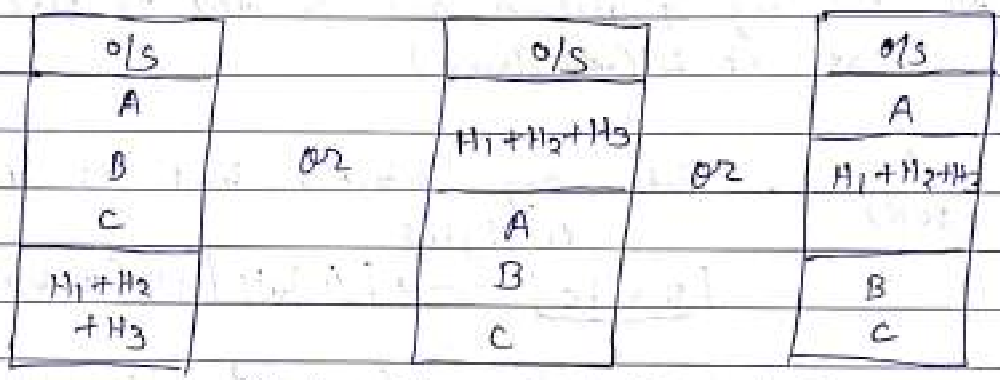
→ no sufficient contiguous Memory.
H_1, H_2, H_3 are scattered

Contiguous (compaction)

$D < H_1 + H_2$
$D < H_2 + H_3$
$D < H_1 + H_3$
$D < H_1 + H_2 + H_3$

if $D > H_1 + H_2 + H_3$, no way we can land D.
so there is no external fragmentation.

when $D < \boxed{H_1 + H_2 + H_3}$ ⇒ Then we have EF = size of D.
↳ size of EF.



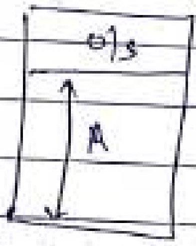
(Relocation of A, B and C)
(and this is called compaction)
↳ (Solution to EF in many cases)

⇒ But Memory Manager do not do this, it attempt to optimize the relocation effort.
~~Memory Manage will always trying to do~~

(3) Table fragmentation (TF)

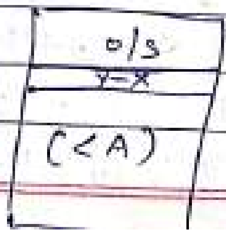
↳ Data structure related to memory management
↳ Used space (contradiction with previous 2 Method)

Memory Management ~~Mem~~ Scheme 1 (TF = X) — Data structure X



Memory M.ment scheme 2 (TF = Y) — Data structure Y

(Y >> X)



(y-x): additional needed space.

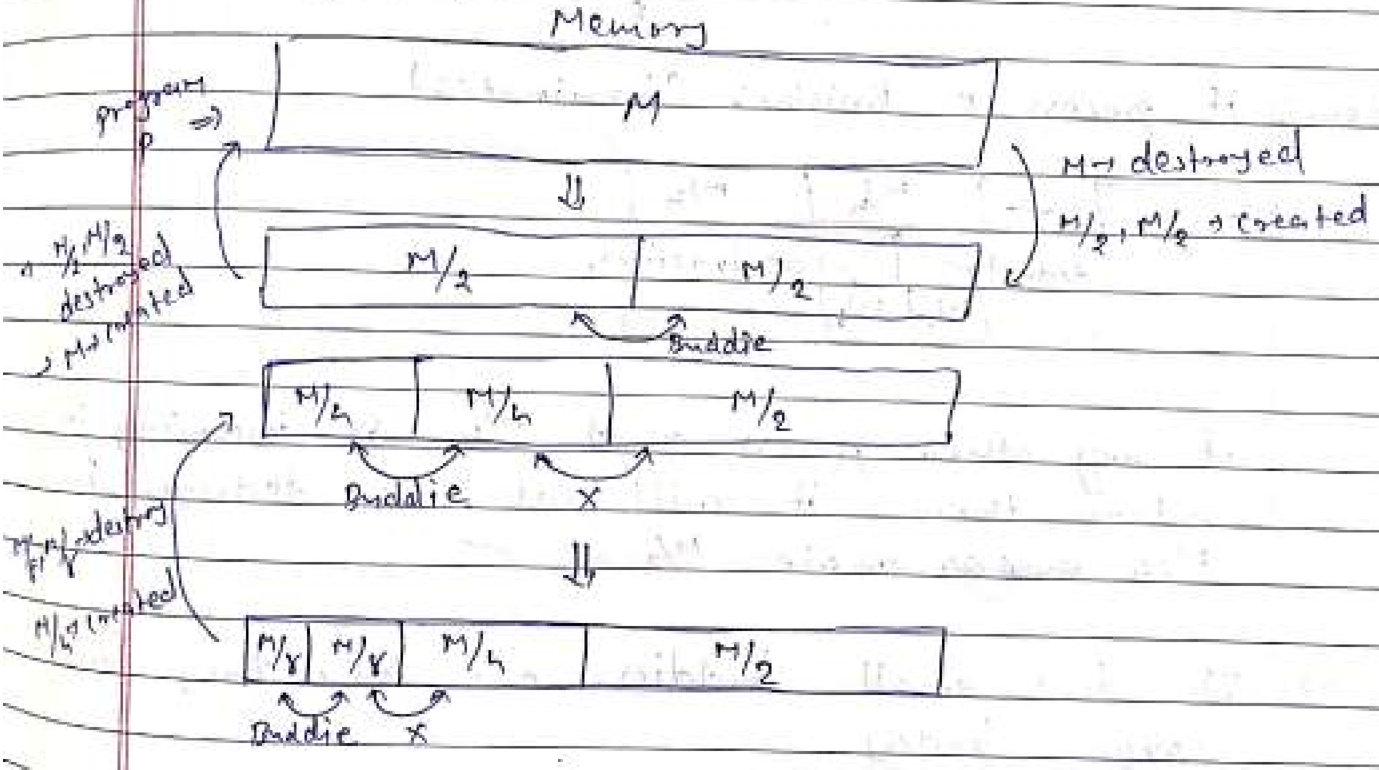
Total fragmentation = IF + EF + TF

⊛ Buddy System (contiguous):

↳ partition (Buddy) created. (Dynamically)

↳ Creation → order (n) (n: division)

↳ Destroyed



⇒ Order: decides minimum size of Buddy.

→ go and creating buddie till buddy of size of incoming program get created.

lets program p : < M/4

then we will stop at M/4 and assign it to p.

Ex: 1 p < M/4 ⇒ till M/4 get created

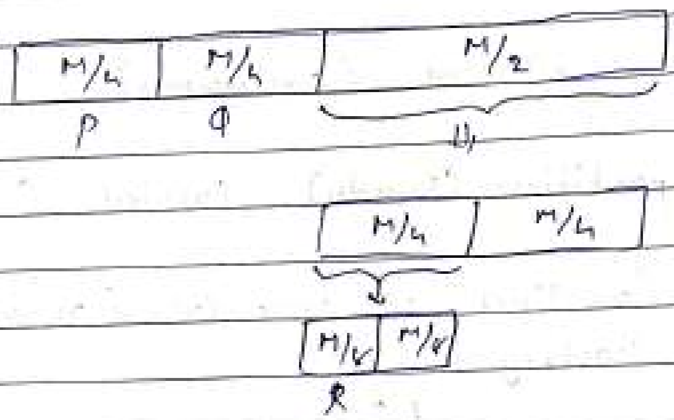
p < M/8 → create M/8 (create M/8 if needed)

(in this case)

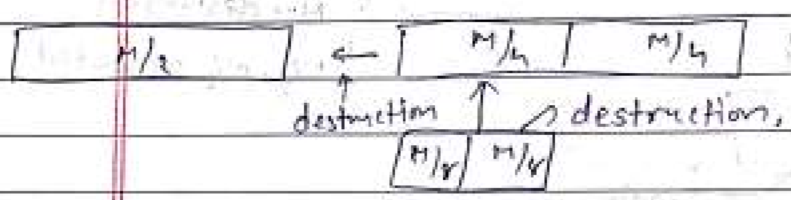
Exa: 2 $P < M/4$ → till $M/4$ get created

$Q < M/4$

$R < M/8$



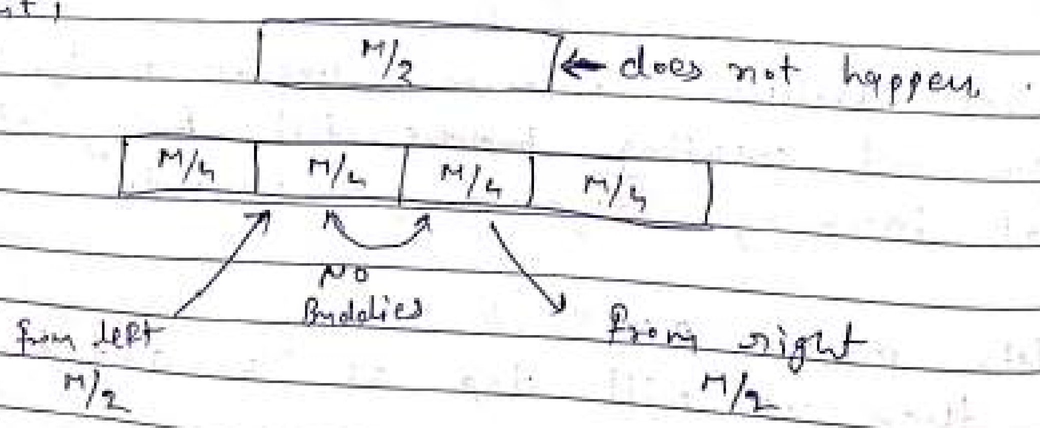
Exa: 3 if process R finishes (terminates)



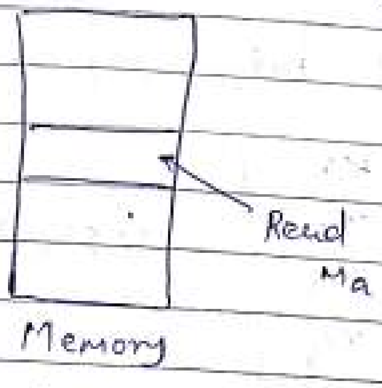
if any other process Z of M/4 incoming in system then it will not get destroyed from M/4 ~~and~~ or create M/2.

⇒ If two small buddies are free, they form a large buddy

But,



⊛ EAT : (Effective Access time) → for Memory Management Technique (MMT)



→ if EAT is close to Ma, the MMT is good.
EAT = Ma (best scenario)

EAT : No of memory access needed to carry out an activity

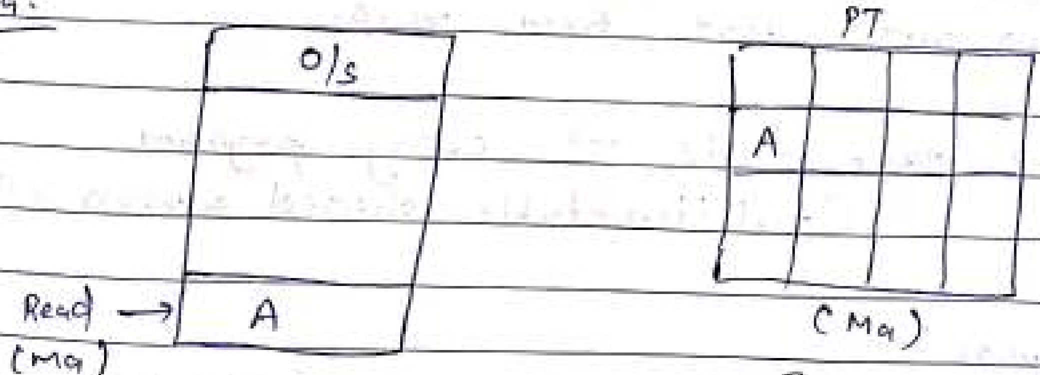
partitioning → Reference to the PT. (in Memory itself)
↓
Reach actual location.

this is also a Memory access (Ma)

also a Memory access (Ma)

so Ma + Ma = 2Ma ⇒ [MFT & MVT)

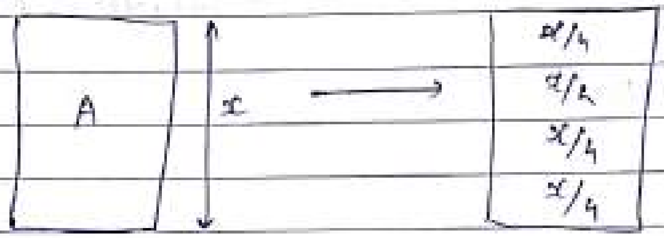
Exg:



2ma
↳ need to reduce this
↳ so we are using paging, ~~...~~ schemes.
↳ need to get closest to Ma.

Paging System: (Non-Contiguous)

↳ program is divided into paging



→ logical space ⟶ physical space
(page) ⟶ (frame or page frame)

→ Size of page and frame is same.
↳ one to one mapping of page to frame

A - 3 pages ⟶ 3 - frames
(0,1,2) (5,9,12)
(contiguous) ↳ non-contiguous)

Page number start from zero. frame number does not start from zero.

↳ page table → every program
(partition-table shared across all programs)

Compare:

Partition & page table
(10 programs)

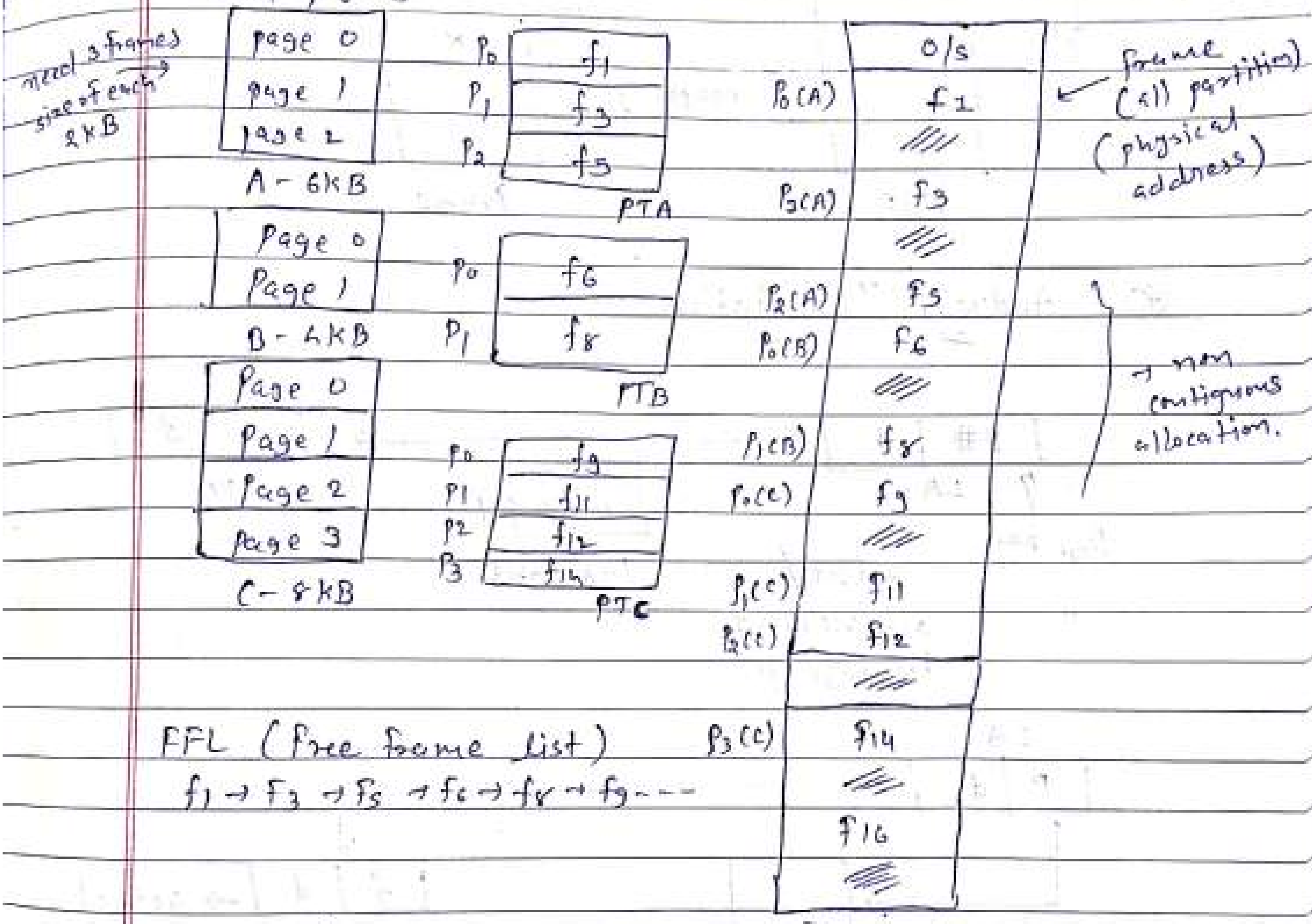
10 entries
(TF - less)

10 page table
(TF - max)

↳ program → spaces

page table → 5 entries
⇓ (PTE)
Total = 5×10 = 50 entries

page size = 2KB
real paging



FFL (Free frame list)
$f_1 \to f_3 \to f_5 \to f_6 \to f_8 \to f_9 - - -$

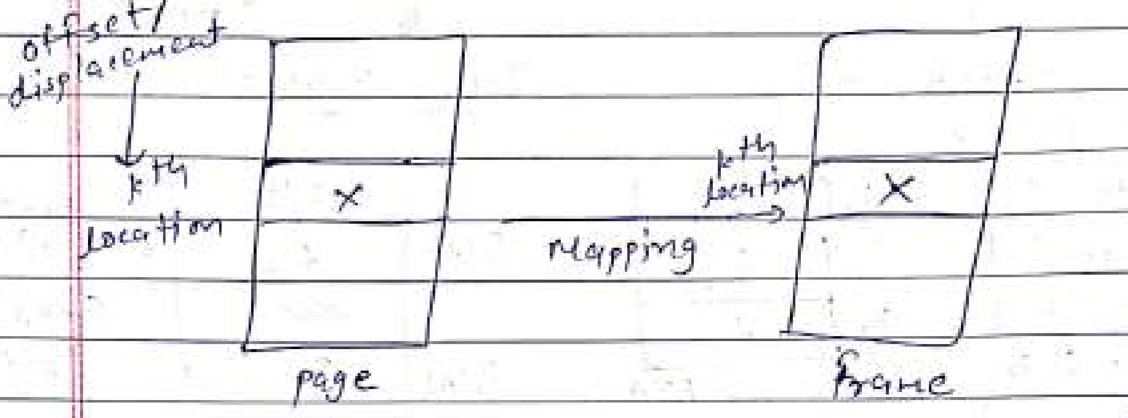
(Mapping page to frame)
(real paging)

⊛ Simple Mapping (Real Mapping):

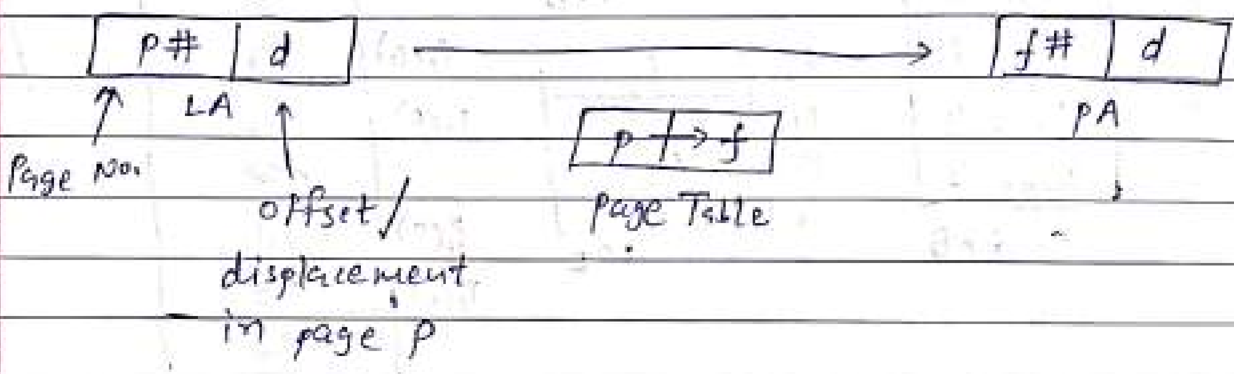
→ Mapping : Page ⟶ frame
(PT : page table)

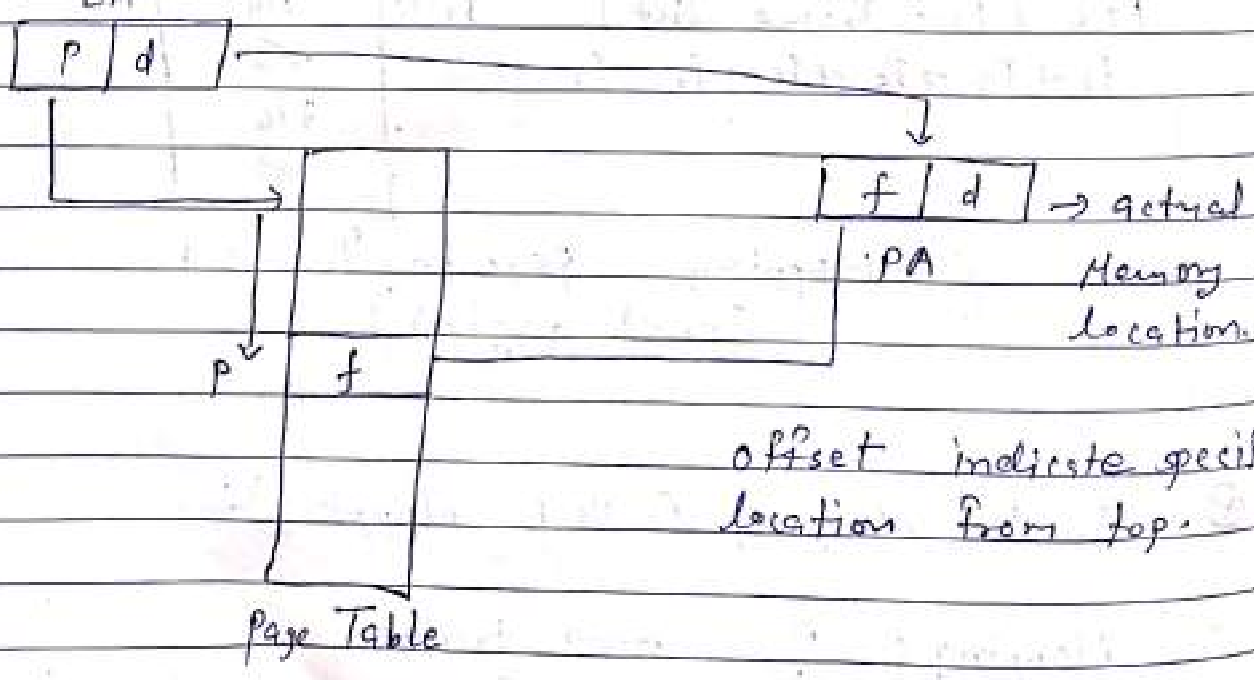
→ Page and frame both of same size
page (2K) ⟶ frame (2K)

page (4K) → ~~frame~~ frame (4K)



✱ Address Translation:





offset indicate specific location from top.

EAT: Effective Access Time

$M_a + M_a = 2M_a$

Page : (can contain) → data
→ stack
→ instruction

Segmentation :→ User / programmer's View

- Code segment → 16K (For Exa.)
- Data segment → 8K (")
- stack segment → 1K (")

→ All segments are variable size.

→ LA ⟶ PA

s#	d

⟶

seg.	d

LA

PA

Segment

offset

Segment Table :

s#	size	

EAT = ma + ma
= 2ma

← all segment are variable size so all time, we need to check size for LA to PA.

⊛→ Size of segment (important parameter)

offset

Address

no. of segments

Segment
↓
Length

8 byte
↓
3 bits
(offset)

64 byte
↓
6 bits (more)
(offset)

Exa:

Program → A → 2 seg. → 0, 1

Program → B → 4 seg. → 2-bits

Segmentation
↳ good (user view)
↳ but complex.

non contiguous:
- Real Paging (simple Paging)
- Real Segmentation (simple segmentation)

→ All pages / All segments } in Memory.

Program A → 4 pages → 4 frames
if 3 frames in Mem. (wait)

Program B → 7 pages

(simple paging)

simple segmentation { if 2 segments in Mem. → Memory

⇒ a complete program or process needed in memory.

- **Page / Segment Table implementation:**

Hardware is always faster as compare to (Page table / Segment table) in Memory.

(1) Hardware → Page Table, Segment Table

Relatively faster

↳ Limited size → limited program size

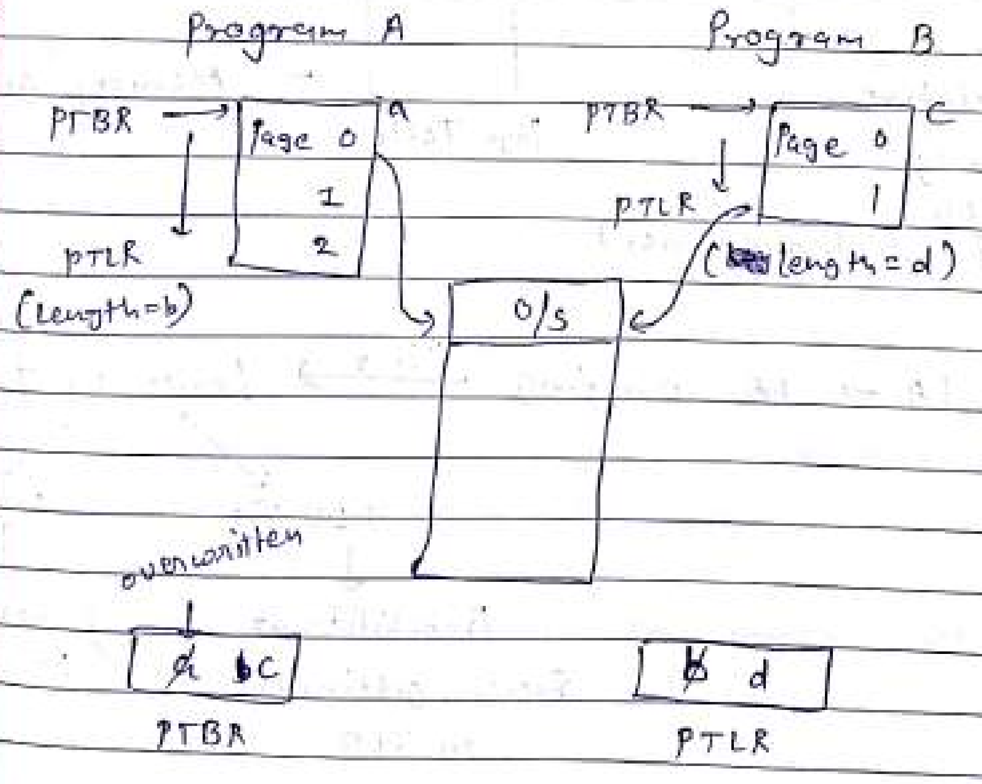
(2) Page table in Memory:

↳ we use specific registers

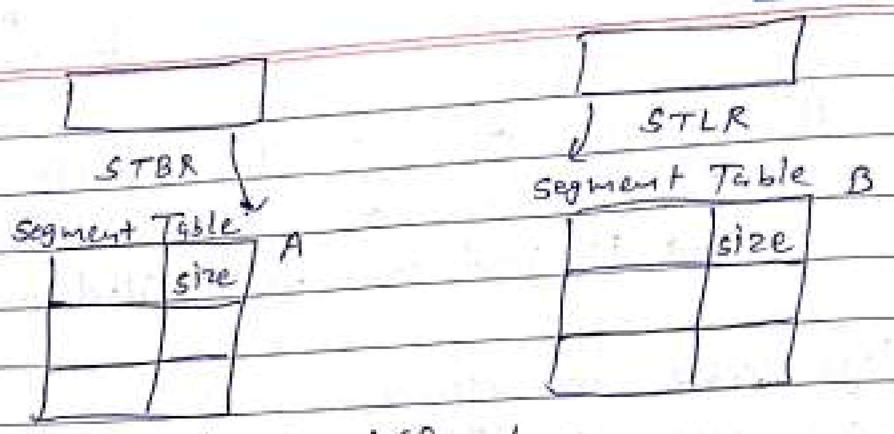
Paging: PTBR (Page Table Base Regi.), PTLR (Page Table Length Regi.)

Segment: STBR (Segment T. Base Regi.), STLR (Segment T. Length Regi.)

Exa:



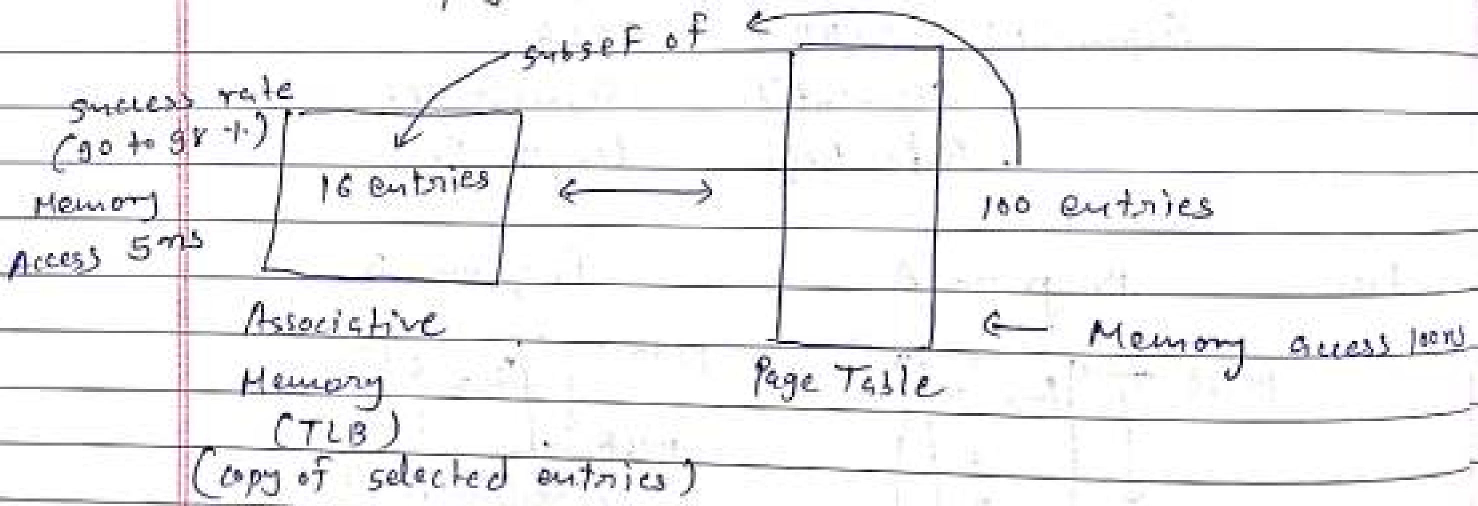
→ Above all is also true for STBR and STLR.



→ All segments size is different
→ Goal is to reduce EAT.

(3) Associative Memory:

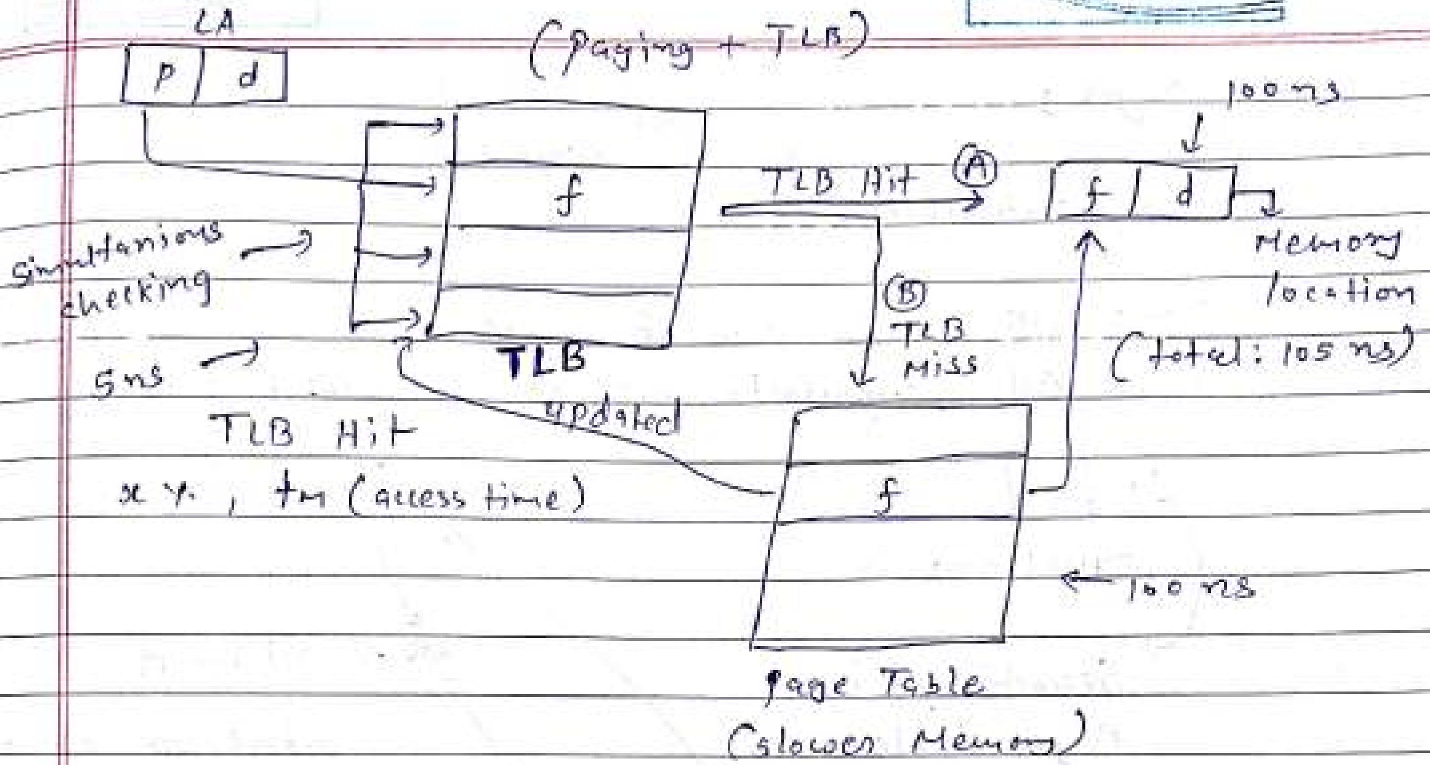
→ TLB (Translation Look aside Buffer)
→ store page table



→ for LA → PA Mapping $\xrightarrow{\text{First}}$ Looks up TLB

- TLB Hit → Probability of Success getting info in TLB
- TLB Miss (Cr=los) → Probability (1 − TLB Hit)

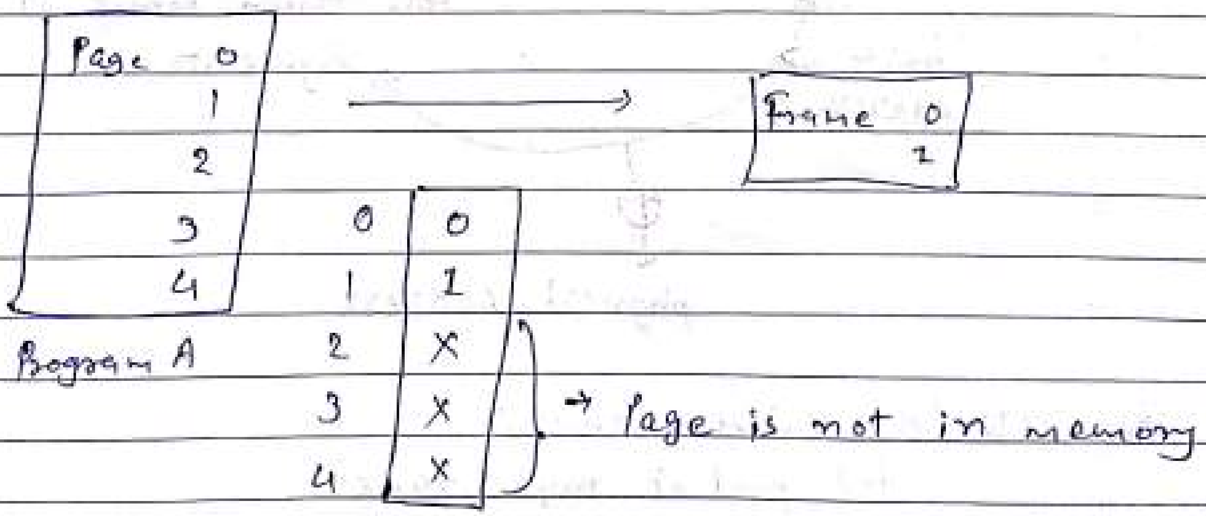
(Paging + TLB)



Path A: 5 + 100 = 105 ns

Path B: 5 + 100 ns + 100 ns = 205 ns

Another Scenario:



→ Running a program in much lesser space.
↳ Virtual Memory.

Scanned by CamScanner

→ Paging : (Real / simple)

Segmentation :

→ All pages are of same size.

→ All segments are of different size

| p | d. |

logical address

→ Searching p in Page table

| s | d |

use → different Value for each Segment

to search Segment in segment table

check that 'd' is within the segment

↓

less than limit of Segment

base → Address

$\oplus$

↓

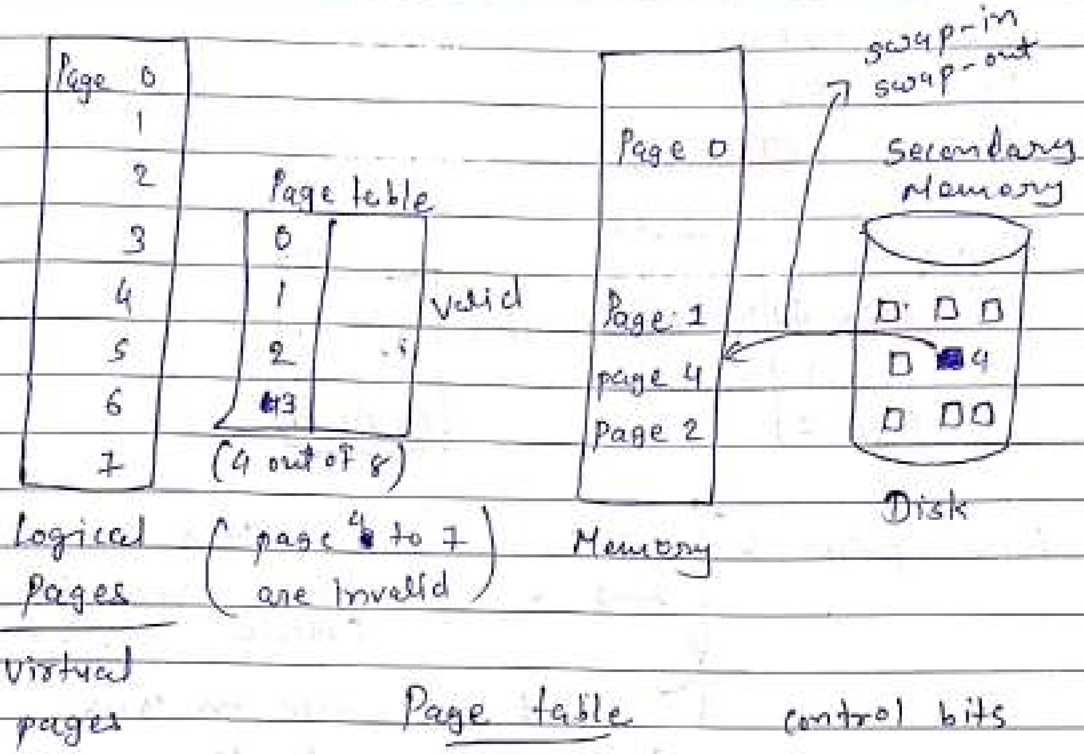
physical Address

⊛ TLB / Associative Memory:

↳ part of page table

⊛ Virtual Memory: → Paging

→ Segmentation.



Page table

control bits

Page #	frame #	

dirty, modify, Read, write, valid / invalid ..

for page 0 to 3 control bit set to 1. (valid)
page 4 to 7 control bit set to 0. (invalid)

→ when Request is trying to access page 4, then page 4 will pull from Disk to Memory.

→ New page that is to be loaded
- ↳ Memory is available : that's ok / NP
- ↳ Memory is not available.

FFL has frames

FFL is empty

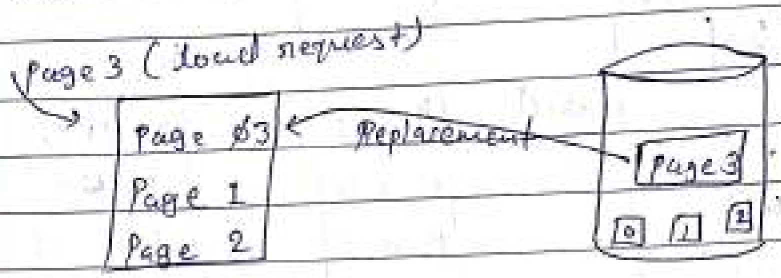
page Replacement
- Global
- Local

+ algos
- FIFO
- LRU
- OPT

LRU : Least Recently Use
OPT : optimal

★ Global FIFO
Local LRU
Global OPT.

Ex:



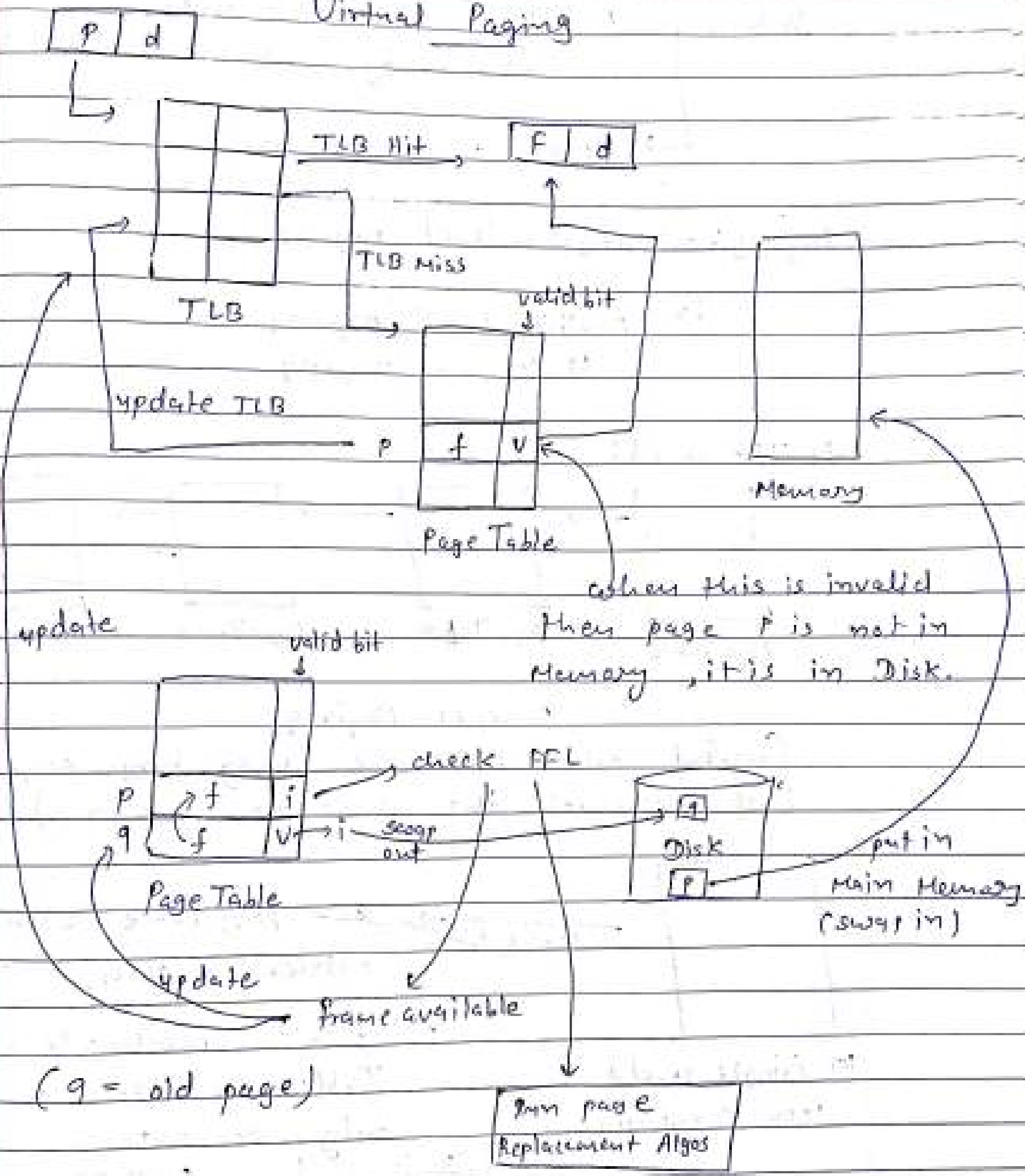
Now page 0 :
- → equal to Disk copy
  - → Just overwrite in Disk
- → different from Disk copy (Made changes in page 0)
  - → copied back to the Disk. (swap out)

→ Copy of page is memory has been written/ Modified after having loaded
↳ Modified page / Dirty page
- → Copy old page back to disk (swap out) — Disk-write
- ↳ bring / load new page (swap in) — Disk-read

Summary

(1) If copy of page in memory is same

Virtual Paging



EAT : Hit + Mis

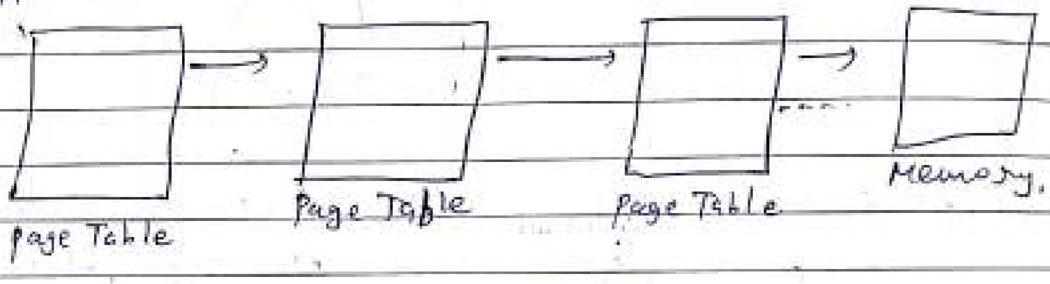
Real Scenario: Hit is 100%.

Virtual Scenario: Hit 80%. Miss 20%.

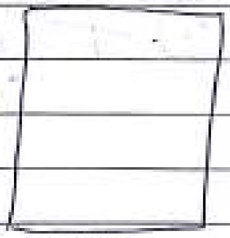
Disk : → x% of pages are dirty
↳ solve by swap out followed by swap in
↳ (1-x) → swap in

Paging : → Single-level paging
↳ Multi-level paging
↳ k-level paging

Multi-level :



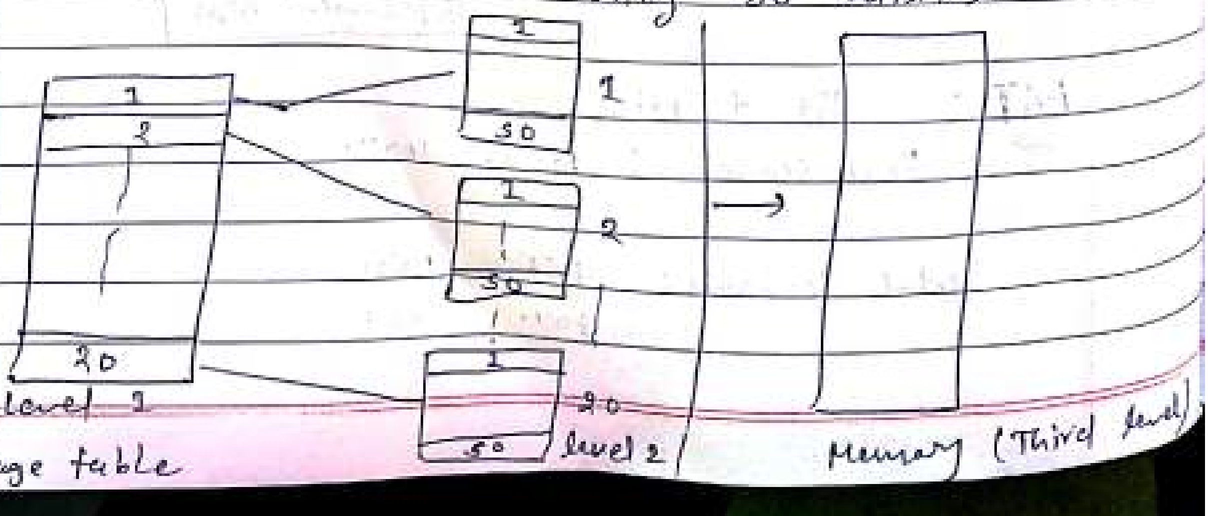
3-level Paging
(useful when we have very large page table)
(it decreases the scope of searching)



PT (single level)
1000 PT entries

$\xrightarrow{\text{convert}}$ 20 - PageTable with 50 entries in each.

→ pointer pointing to each Table so need to search only 50 entries.



SAKHEE
Date: / /20
Pg.No :

EAT ⇒ $\boxed{(K+1)\ ma}$

↳ (K-level paging) and one for Memory

* Replacement Algos!

↳ Local : process specific

↳ Global : Across all processes.