17/03/25

```
signal (semaphore *s) {
    s -> value ++;
    if (s -> value >= 0) {
        remove process Pi from S -> list;
        wakeup (P); // bring the process back
                    // to ready queue
    }
}
```

Deadlocks & Starvation

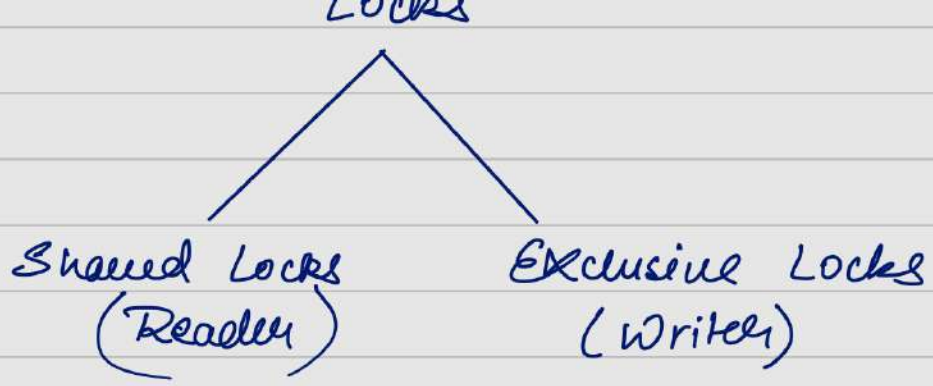
P_0	P_1
wait(s);	wait(q);
wait(q);	wait(s);
⋮	⋮
signal(s);	signal(q);
signal(q);	signal(s);

Readers - Writers Problem

→ Database D, 'n' users

↓ read / query
write / modify

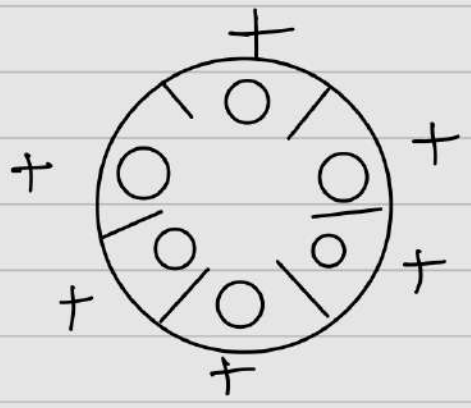
Read
Write ← exclusive



→ If a writer is waiting, then no new reader should be allowed.

☆ Priority of writer > reader

Dining - Philosopher's Problem



18/3/25 Deadlock

System :-

→ Different H/W Resources

→ Disks
→ Printers

} Finite fixed no. for a given system

Users ⇒ accessing the system

↳ Request the resources

Task → grant access to the resources → no deadlock

Users

1. Request for a resource
   ↓ Resource granted
   Use the resource
   ↓ on completion
   Release the resource

Four Conditions for deadlock

1. Mutual Exclusion

319. Galvin

Some resources will always be in non Shareable mode

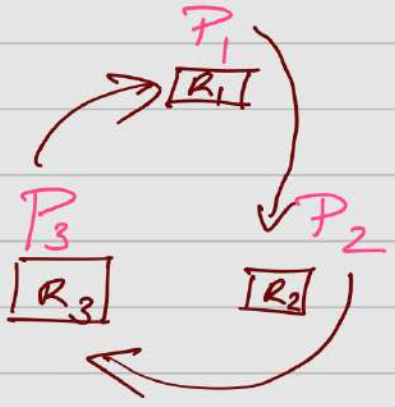
2. Hold and Wait

P_1 is holding a resource R_1 and waiting to acquire resource R_2.

3. No Pre-emption

A process releases resources voluntarily only after completion.

4. Circular Wait



P_1 is holding R_1
P_2 " " R_2
P_3 " " R_3

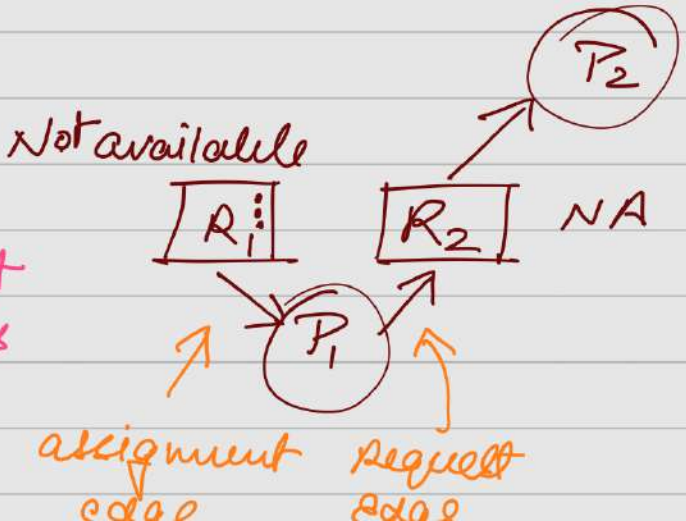
P_1 has requested for R_2
P_2 " " " R_3
P_3 " " " R_1

Resource allocation Graph

1. Request Edge $P_i \rightarrow R_j$
2. Assignment Edge $R_j \rightarrow P_i$

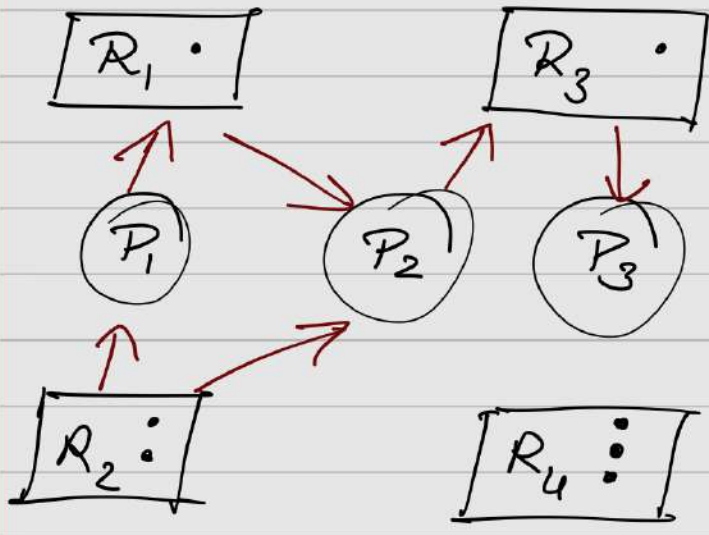
A.E => The resource has already been assigned to a process

R.E => The process has requested for this resource



One unit of R_i has been assigned to P_1.

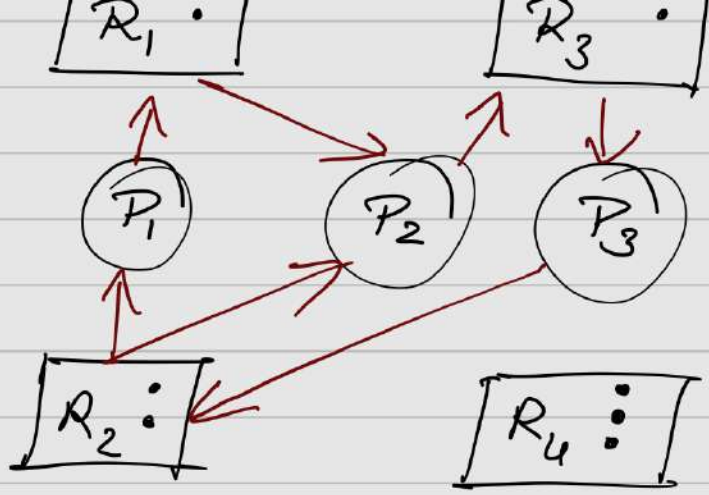
P_1 is requesting for R_2. R_2 has already been assigned to P_2.
$\therefore$ P_1 is in hold & wait.



P_2 is holding R_1
P_3 is holding R_3

Both P_1 & P_2 are holding R_2

If there are no cycles in the graph then there is no deadlock in the system. However if there is a cycle then deadlock may exist

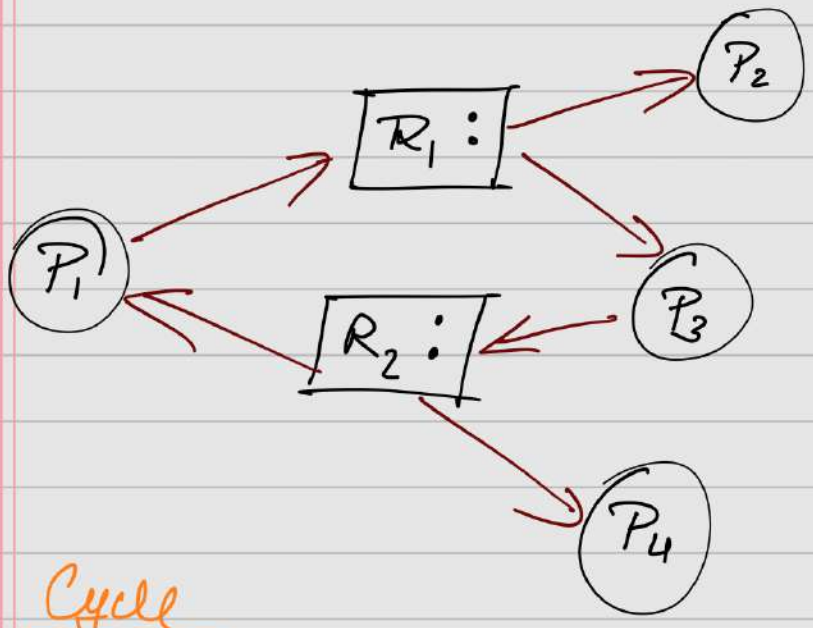


Circles

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Unsafe deadlock



Cycle

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

No deadlock

P_4 is not satisfying hold & wait condition.

Methods for handling deadlock

- Deadlock prevention
- Deadlock avoidance
- Deadlock Detection

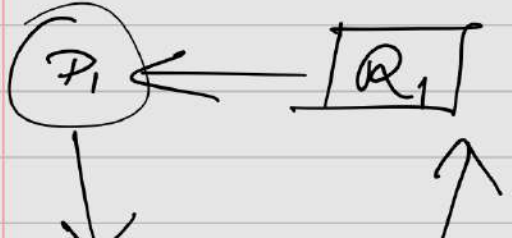
19/5/25

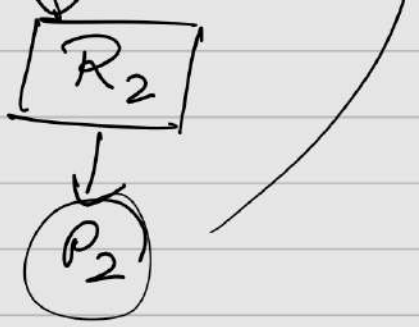
Deadlock prevention - find strategies to ensure that atleast one of the necessary conditions do not hold.

- Mutual Exclusion ⇒ Resources are held in shareable mode only (practically difficult)
- Hold & Wait ⇒ whenever a process requests resources, it doesn't hold any other resources (This condition has to be ensured)

↳ (2 disk, one printer) blocked for other process

↓ read write } less throughout } leads to starvation for other processes





$P_1 \Rightarrow$ Both R_1 & R_2 to complete

P_1
↓
Asking for R_1
set of instructions

Asking for R_2 → System checks → If R_2 is unavailable

R_1 is released

↓

Pre empts P_1 and releases all resources of P_1

Deadlock prevention is not always possible

Deadlock Avoidance

⇒ Safe sequence

Multiple processes

Requesting multiple resources

of different types, where each resource may have multiple units.

Safe state

A state is safe if system can allocate resource to each process in some order and still avoid a deadlock.
The sequence in which these process are allocated resources leading to safe state is known as safe sequence.

R = 21 units

Five processes requesting 5 units each

P_1	P_2	P_3	P_4	P_5	
4	4	4	4	4	= 20

R = 18, Six processes requesting 4 units each

P_1	P_2	P_3	P_4	P_5	P_6	
3	3	3	3	3	3	= 18

R = m units

n processes requesting k instances each

n_2 " " k_2 " "

$1, 2, \ldots n_1$ $1, 2, \ldots n_2$

$(k_1-1)\ (k_1-1)\ \ldots\ (k_1-1)$ $(k_2-1)\ (k_2-1)\ \ldots\ (k_2-1)$

Safe state = $m > n_1(k_1-1) + n_2(k_2-1)$

system specific

Bankers Algorithm for deadlock Avoidance

1. Available → no of available resources of each type in the system

2. Max → maximum demand from a process
   (P_1) → 2 units (R_1) 1 unit (R_2)
   after 1 hour
   1 unit (R_1)
   Total
   3 units (R_1)
   1 units (R_2)

3. Allocation → no of resources of each type currently allocated to a process

4. Need → Max − allocation

Remaining no of resources that the process might ask for at some point of time

process

specific

Available ⇒ [X, Y, Z] = [5, 5, 5]

	Allocation			Need		
	X	Y	Z	X	Y	Z
P_0	1	2	1	1	0	1
P_1	2	0	1	0	1	2
P_2	2	2	1	1	2	3
	5	4	3	2	3	6

If it is in safe state & what is safe sequence?

Free ⇒ x = 0
y = 1
z = 2

[0 1 2]

P_1 completed first

Free ⇒ x = 2
y = 1
z = 3

P_0 completed

Free ⇒ x = 3

Y = 3

Z = 4

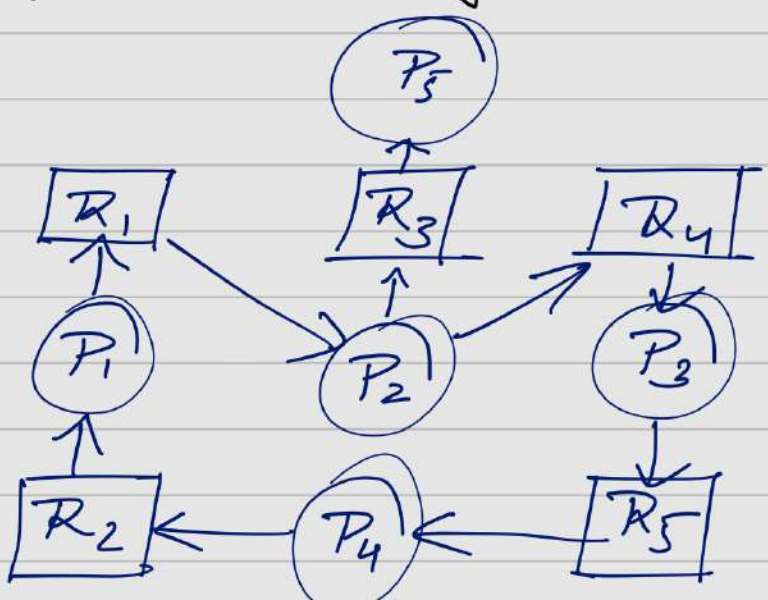
P_1, P_0, P_2 ⇒ Safe sequence

24/03/25

Deadlock Detection

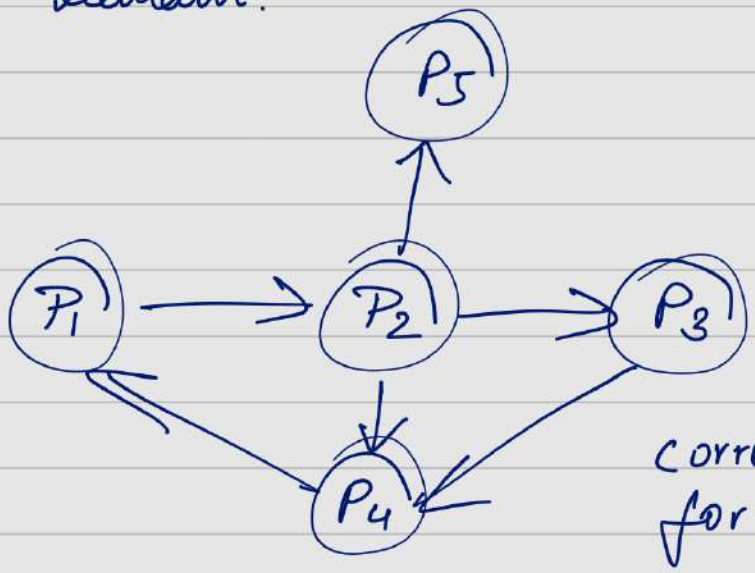
- Detect whether there is a deadlock at all (repeatitive)
- If you find a deadlock
  - Solve it (prevention)
  - (Recovery from deadlock)
- Resources with one instance
- " " multiple "

Single Instance of Each Resource Type



Resource allocation Graph

Weight for Graph
which states that we will dissolve all resources and edges will remain.



corresponding wait-for graph

$P_i \rightarrow P_j$ means that P_i is waiting for P_j to release a resource held by P_j now, which is needed / requested by P_i.

Check whether cycle is present ?

The deadlock detection algo is repeatedly runn. (n^2)

Recovery

Process Termination

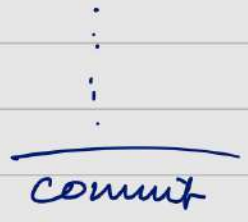
Resource Pre-emption

→ Abort all processes involved in the deadlock
→ Abort one process at a time, until cycle is removed.

→ victim ?
→ Rollback
→ Starvation

Aborting ⇒ forcefully terminating a process when its running

Victim :⇒ whichever process that is selected to be preempted.



$S_1 \rightarrow sum = sum + S_1$
$S_2 \rightarrow sum = sum + S_2$
S_3 ⋮ ⋮

Rollback ⇒ When the process (victim) goes back to a previous

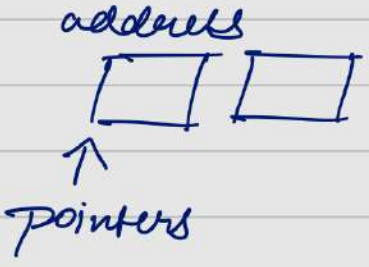
state (safely).

Starvation → When the process is always preempted or it never gets the chance to execute. (P_4 in this case).

Module 4

Memory Management

- Main Memory
- Virtual "

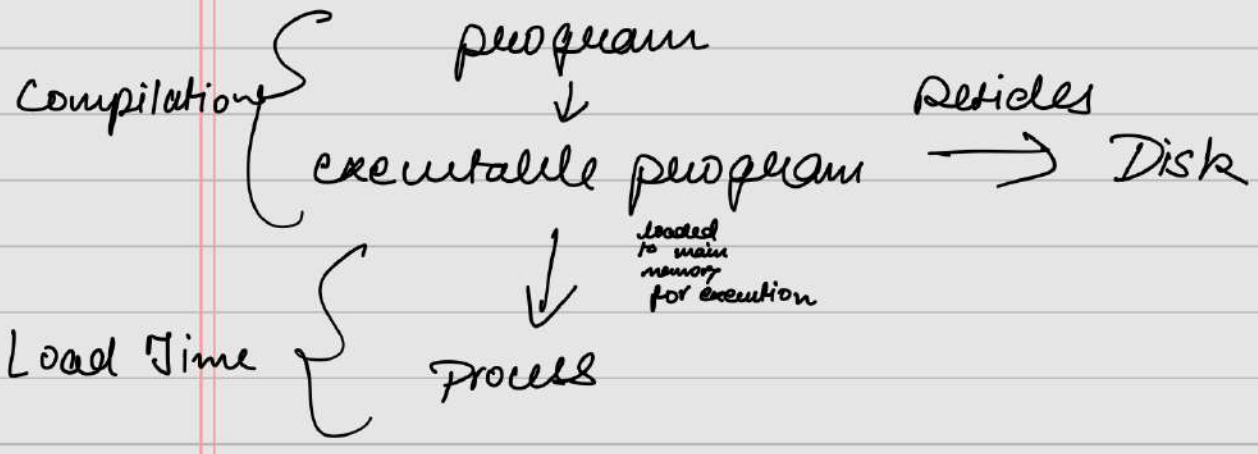


Store that in our disk. (program) —Execute / Run→ Main Memory (process)

Rel^n b/w programs & process addresses

25/03/25

- Compile Time
- Load Time
- Execution Time



Address Binding

Activity maps your program to one address that

- If we know where our executable is at the disk, we can perform address binding at compilation. (less chance)
- When we know the address in main memory, address binding is performed at load time. (most common)
- Special case - we stop a running

process, then map it address.

Logical Address Vs. Physical address

Logical Address Space: CPU generates an address → Logical Address

Physical address space: Corresponding address seen by main memory ↓ Physical memory

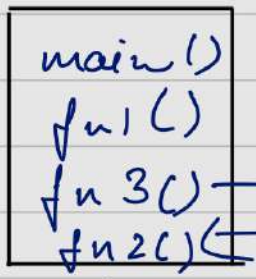
If A.B is done during C.T or L.T, then logical address = Physical address.

Reason - In execution time, the process is preempted, ∴ its logical address is different than physical address.

Dynamic Loading

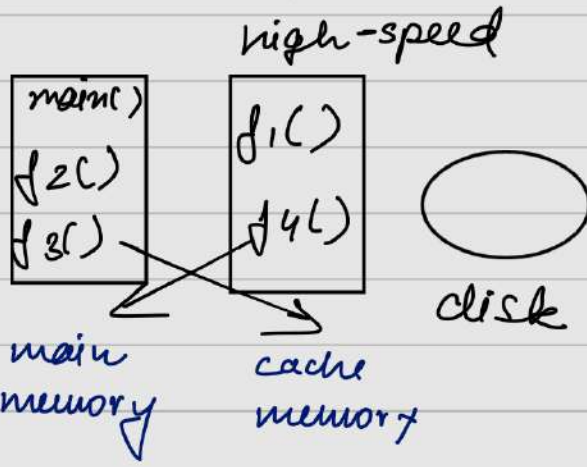
- when entire program can't be loaded into main memory.

routines



overhead

Swapping



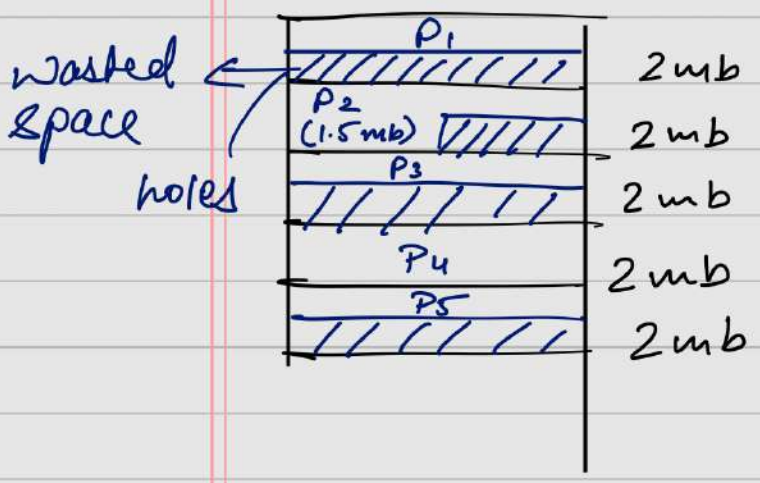
5 functions

while executing, if we need a part of program which is not in main memory but on disk. We remove non vital part with the required part from the disk.

The overhead in swapping is much less compared to dynamic loading because we swap from cache, which is faster to access.

Contiguous Memory Location

- fixed size partitioning
- variable-size partitioning



P1 - 1mb

P4 → 2mb

1 + 0.5 + 1 + 1 = 3.5 mb

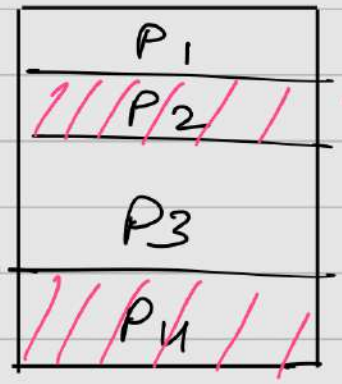
- Internal fragmentation

↓

compaction (Rearranging process

so extra space is shifted at last so it can be used by other process)

Variable - Size Partitioning



P_1 (5mb)

P_2 (2mb) → 2mb (free)

P_3 (4mb)

P_4 (1mb) → 1 mb (free)

P_6 (1mb)

Algo to decide where to put a new process in case of multiple free space.

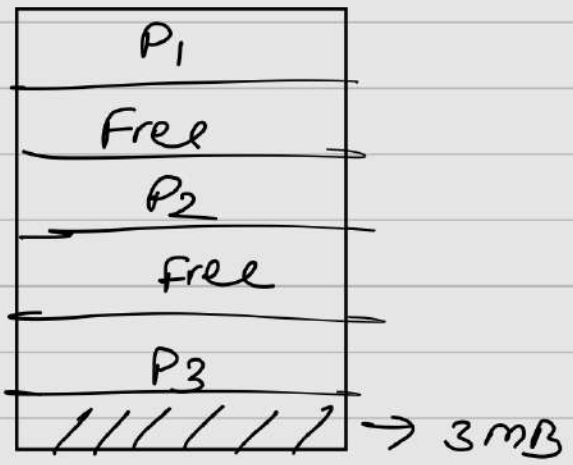
1. First fit → starts where previously ended [from beginning during initial call]
2. Best fit
3. Worst fit

26/3/24 Variable Size Partioning

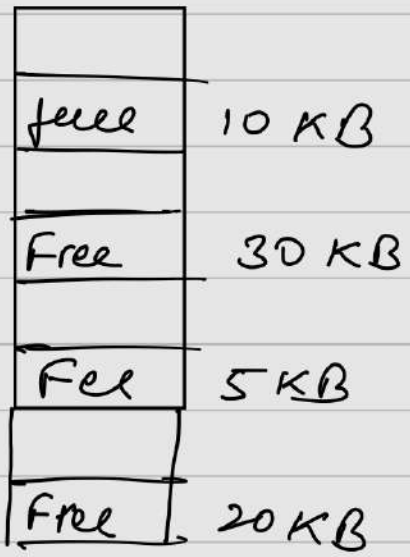
- External fragmentation

↓ solution

Compaction



Q1.



Show allocation for $R_1 = 20$ KB,

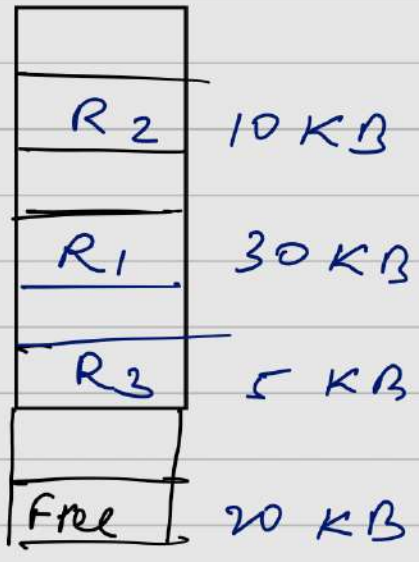
$R_2 = 10$ KB

&

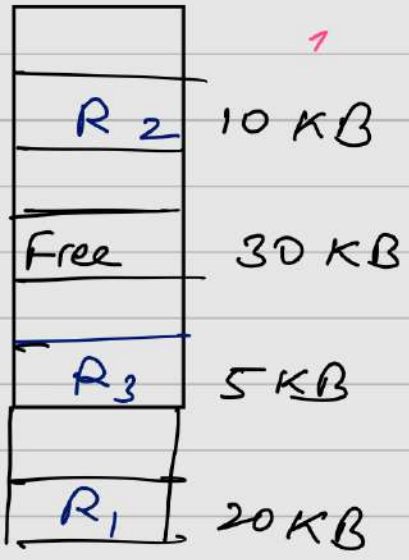
$R_3 = 5$ KB

Using First, Best & Worst fit.

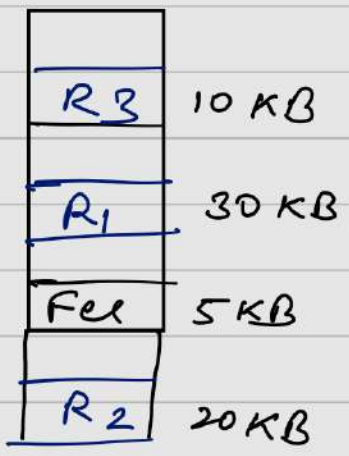
First fit



Best fit



Worst



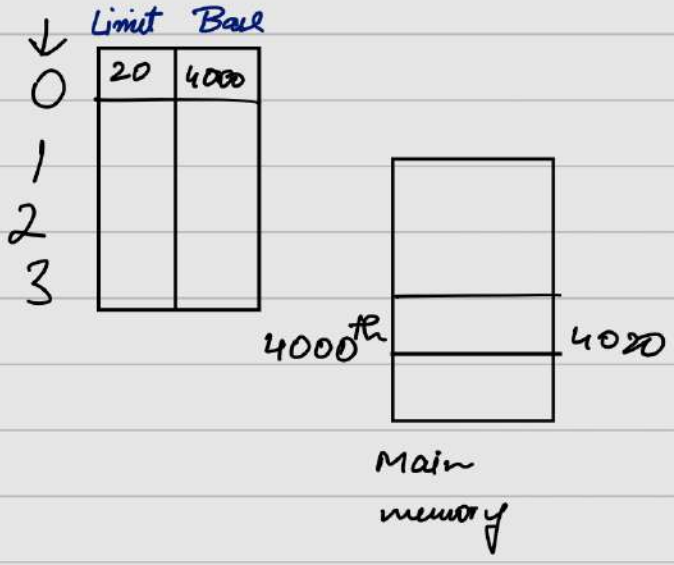
Also

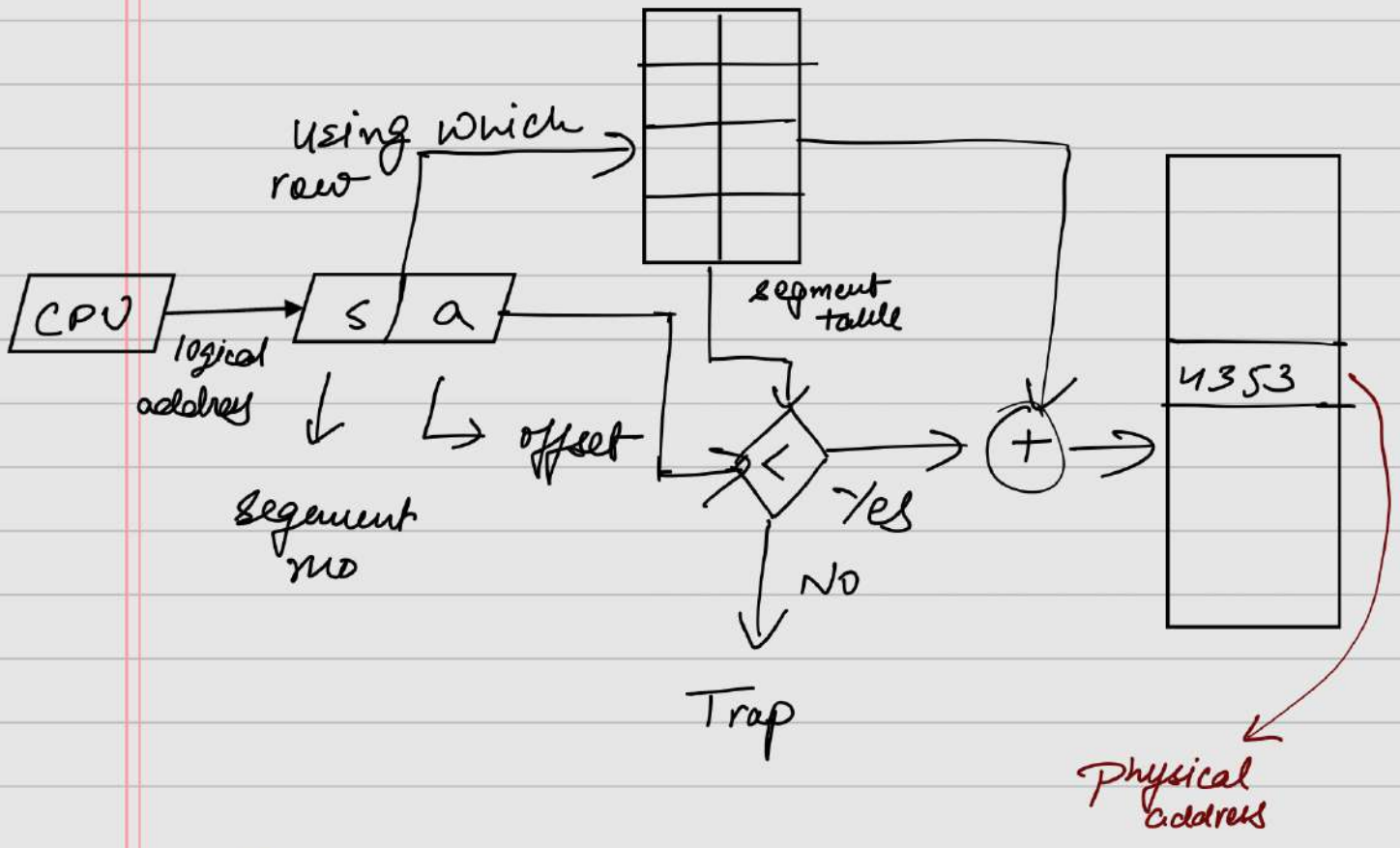
Non - Contiguous Allocation

- Segmentation
- Paging

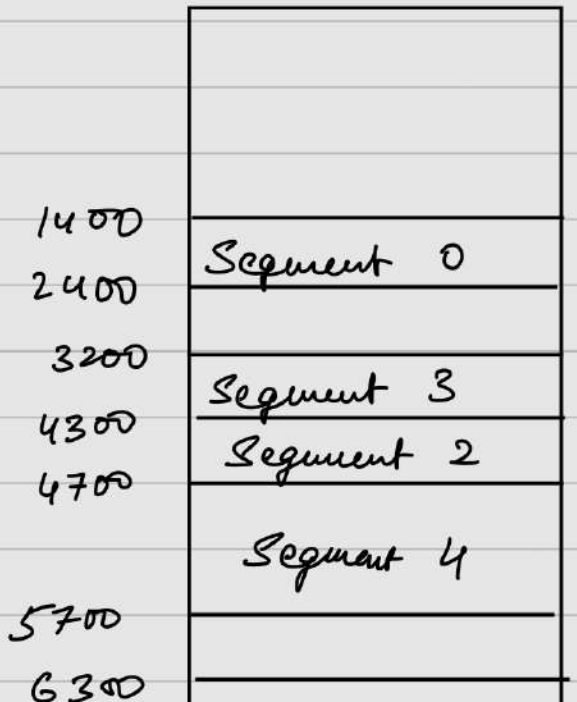
Segmentation

Segment Table





If limit is less than offset, go to trap



	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3000
4	1000	4700

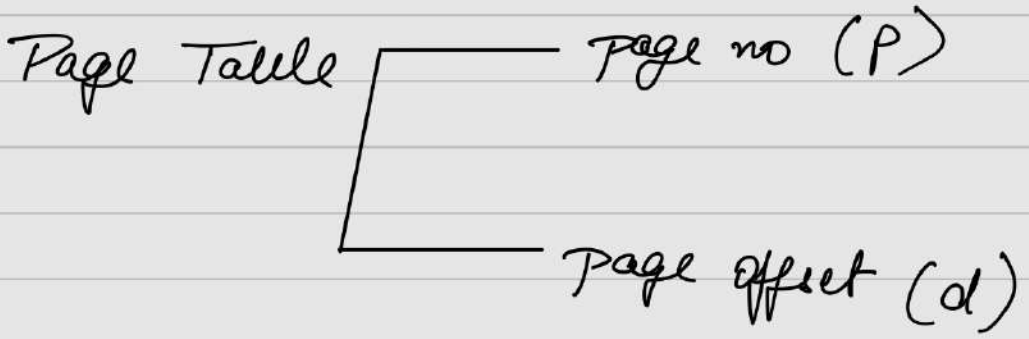
Segment table

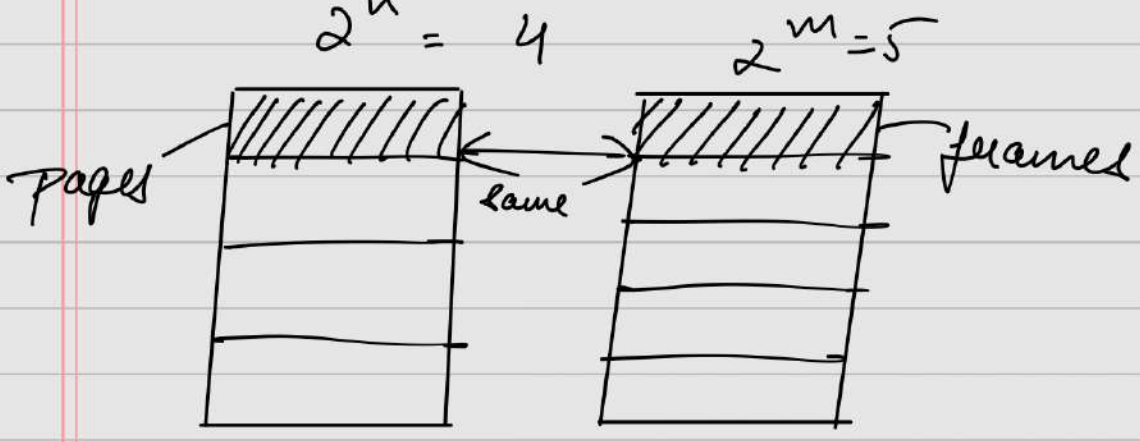
2/4/25

Paging

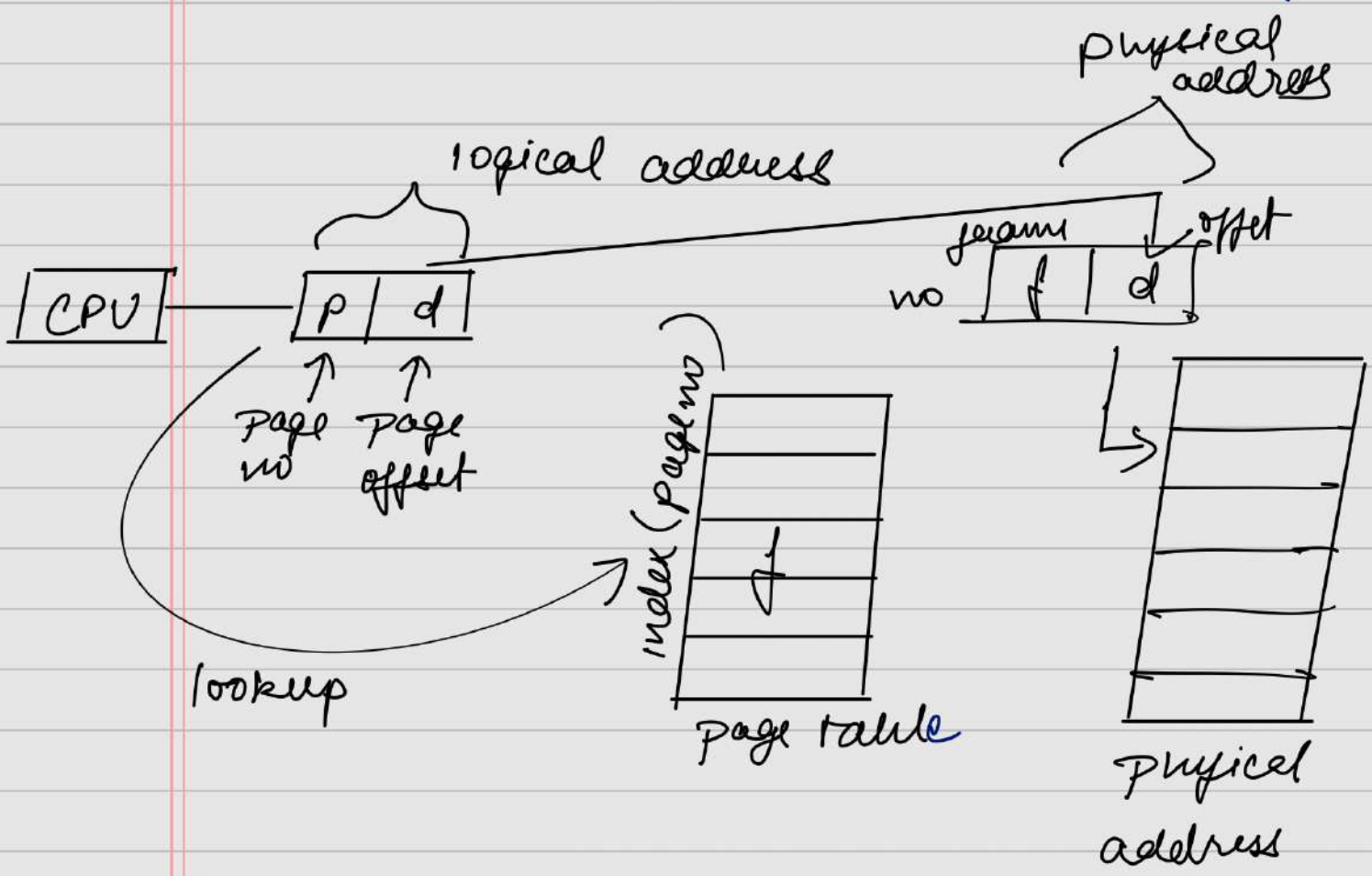
Physical memory is broken into fixed-size blocks ⇒ known as frames

Logical memory is broken into same size pages.



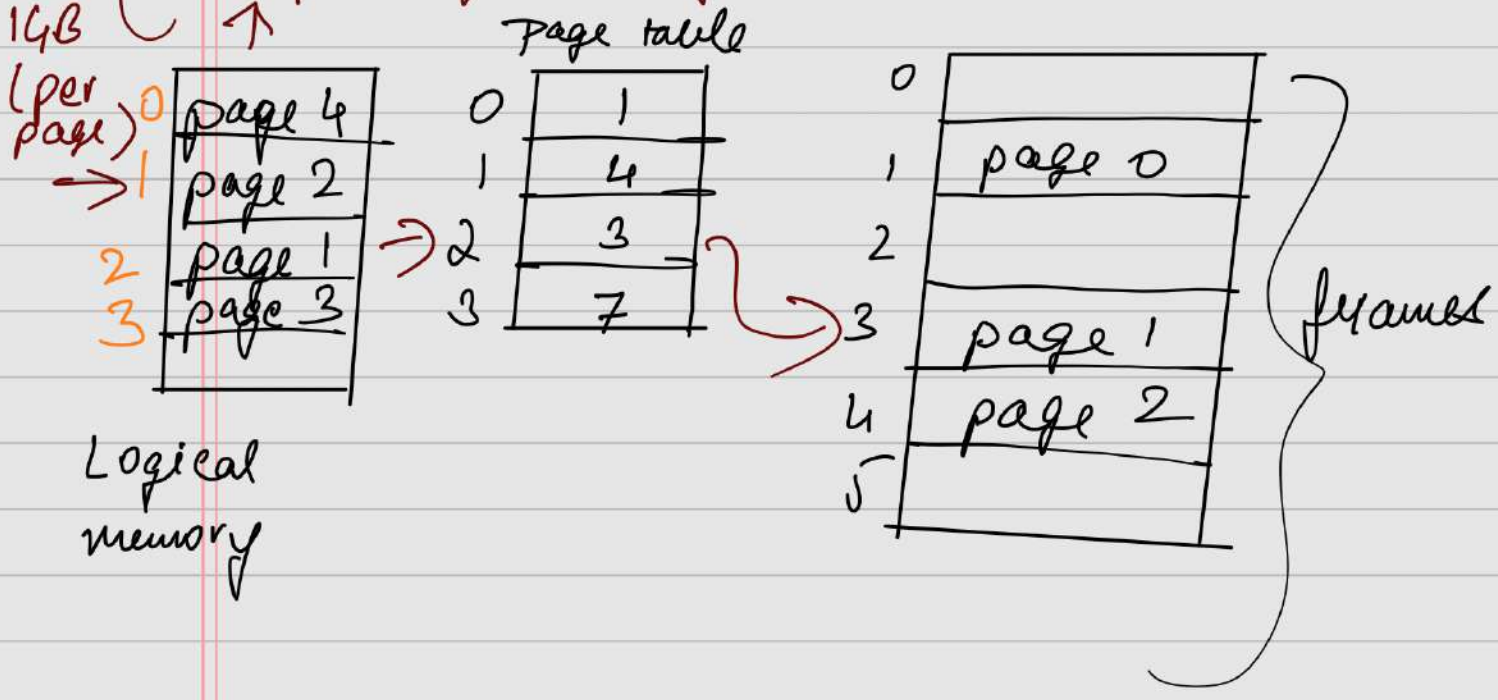


- Logical memory > Physical memory



Paging

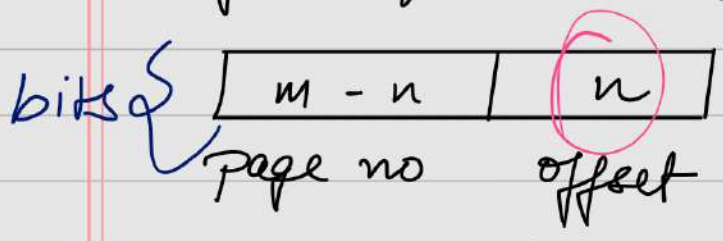
512 MB to 1GB (per page) } Size is always in power of 2



Assume

Size of logical address = 2^m

Page size = 2^n bytes



looking for a

$(5 \times 4) + 0$
$\Rightarrow 20$

logical memory → 16 (0-15)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

page 0 (4, 0-3)

page 1

page 2

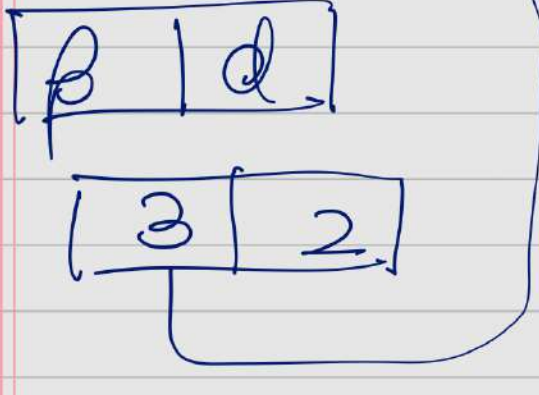
page 3

page table

0	5
1	6
2	1
3	2

0	
4	i j
8	k l
12	m n
16	o p
20	
24	a b
28	c d
32	e f
	g h

4



$(6 \times 4) + 1$

$\Rightarrow 25$

$m = 4$ → $2^4 = 16$ size of Logical address space

$n = 2$ → $2^2 = 4$ size of each page

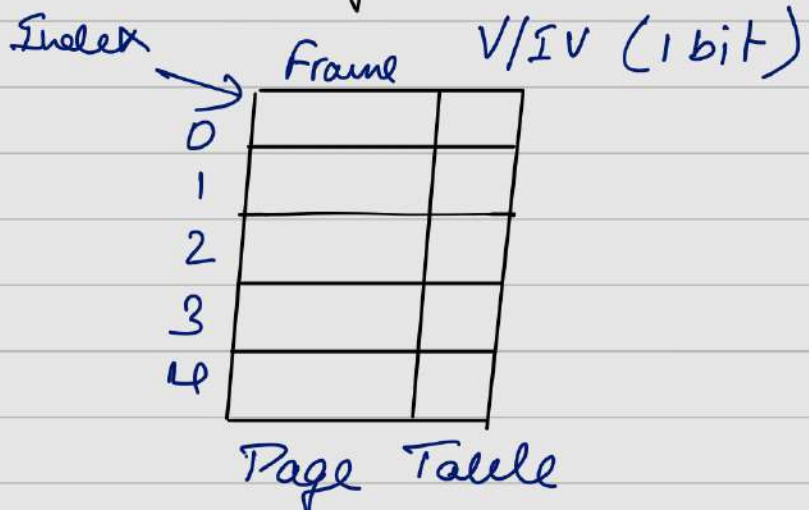
$\frac{16}{4} \Rightarrow 4$ (no of pages)

8|4|25

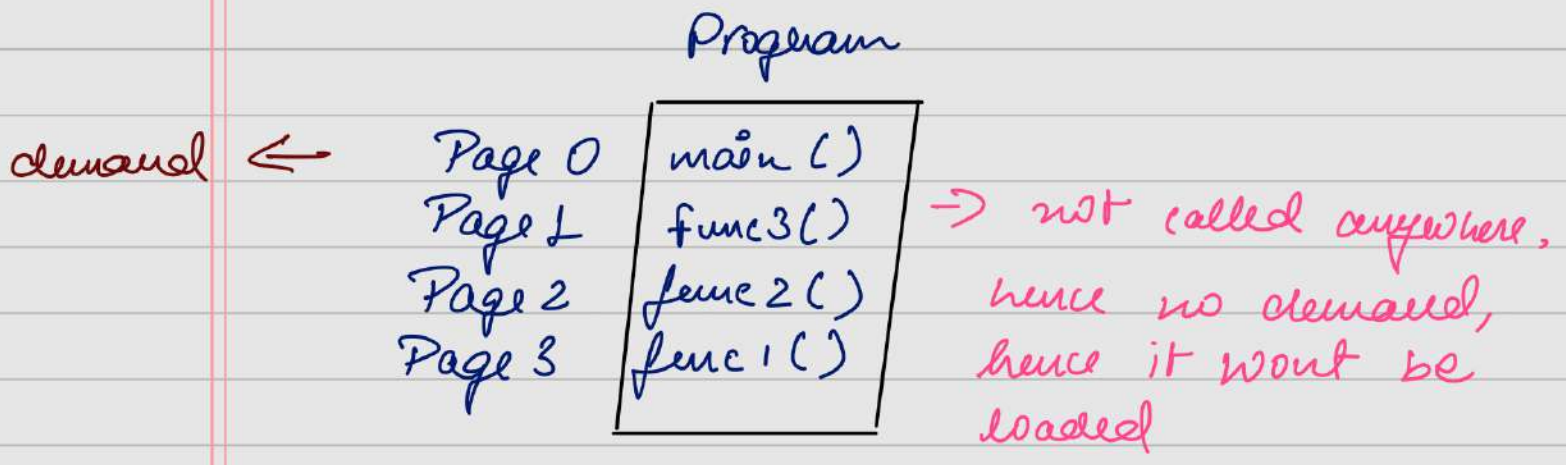
Virtual Memory

Seperation of logical memory as perceived by users from that of the Physical memory.

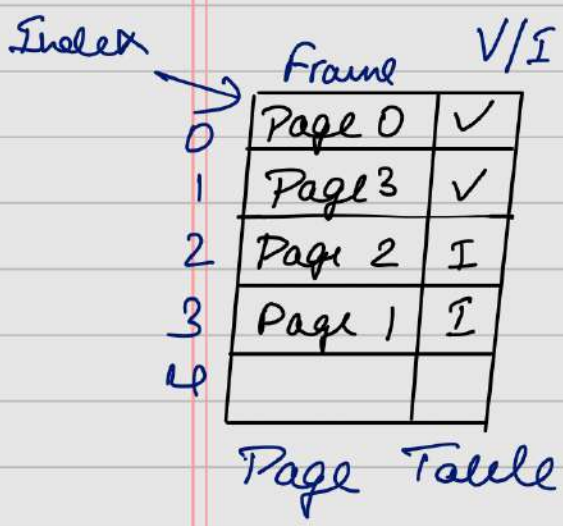
Demand Paging



When there is a demand for a particular page, we will bring that page to the main memory.



The technique of loading in pages only when it is needed during prog execution.



Page 0 ← main memory

Page 3 () ← func () ← main memory

```
main()
{
    func1()
    .....
    func1()
}
```

Page fault → when a page which is needed is invalid in page table.
Initially all pages are invalid.
(Pure demand paging)

Effective Memory Access

"ma" → memory access time
"p" → probability of a page fault

$$EAT = p \times \text{fault time} + (1-p) \times ma$$

↓
time required to bring the page from secondary memory

fault-time = 8 ms
ma = 200 ns
p = 0.25

3 different Page Replacement Algo

- FIFO
- Optimal
- Least recently used (LRU)

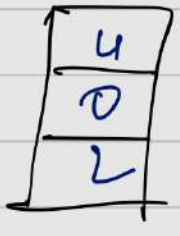
→ Victim page ⇒ remove / swap out that page from main memory to bring in / swap that new page.

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

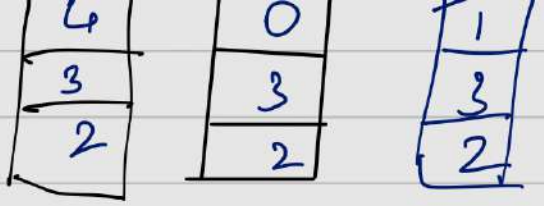
FIFO →

Look through the entire sequence & choose the least used page

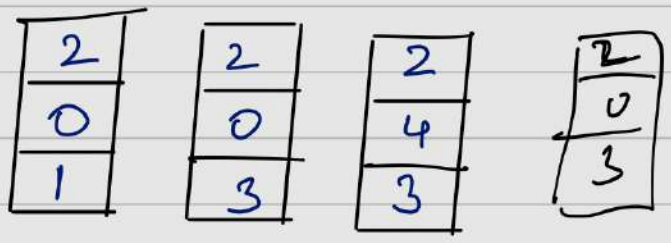
LRU ⇒



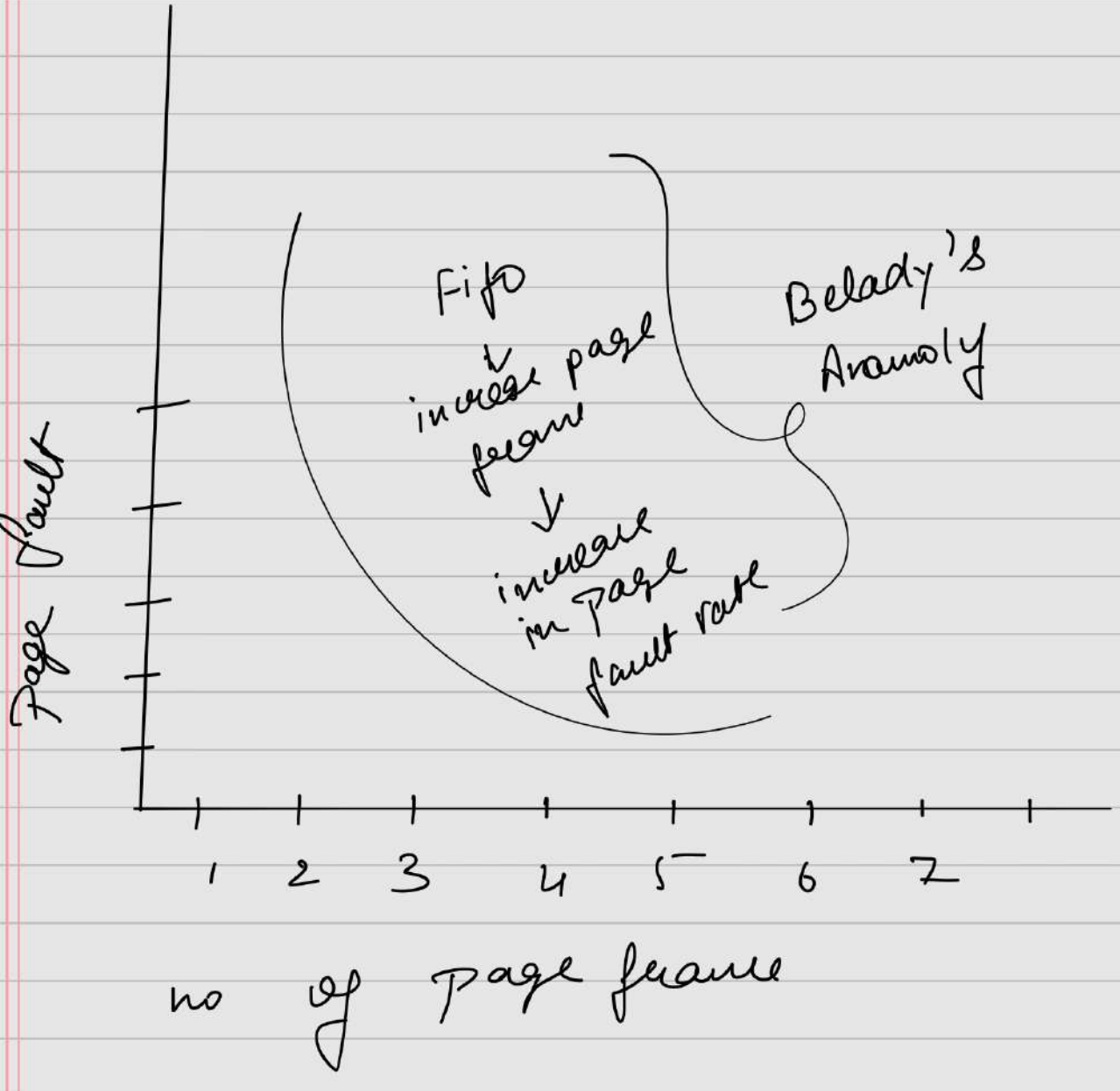
7 was the last one used



Optimal →



FIFO in case of tie



Disk Management

Thrashing :⇒ This high paging activity is called thrashing

Paging activity :- Page fault occurs, so you have to bring in new pages.

$$(1-p) \times \text{fault rate} + p \times ma$$

Spending more time in doing paging related tasks rather than executing

Magnetic Disks ← Secondary Storage

Access Time

Seek time → TR to
Rotational latency →
Transfer rate ⇒ rate at which data flows from the disk to the computer

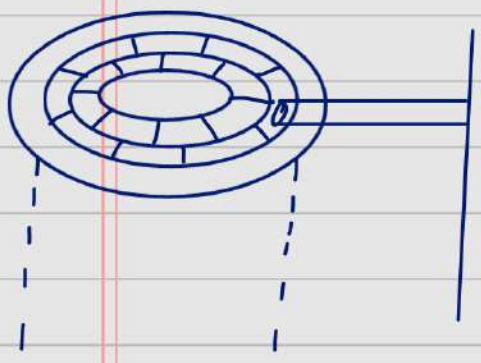
Random Access Time ⇒ Seek Time + Rotational latency

Seek time - Time taken to remove the disk arm / RW head to the desired cylinder

15/4/25

Rotational latency - Time needed by

disk head.



Disk drive is attached to a computer with a set of wires :- known as (I/O Bus)

- → ATA (Advanced Technology attachment)
- → Serial ATA (SATA)
- → eSATA
- → Universal Serial Bus (USB)

Bandwidth :- the total number of bytes transferred, divided by the total time b/w the first request & the completion of last transfer.

(3 marks)

Disk Scheduling [1 Question from this]

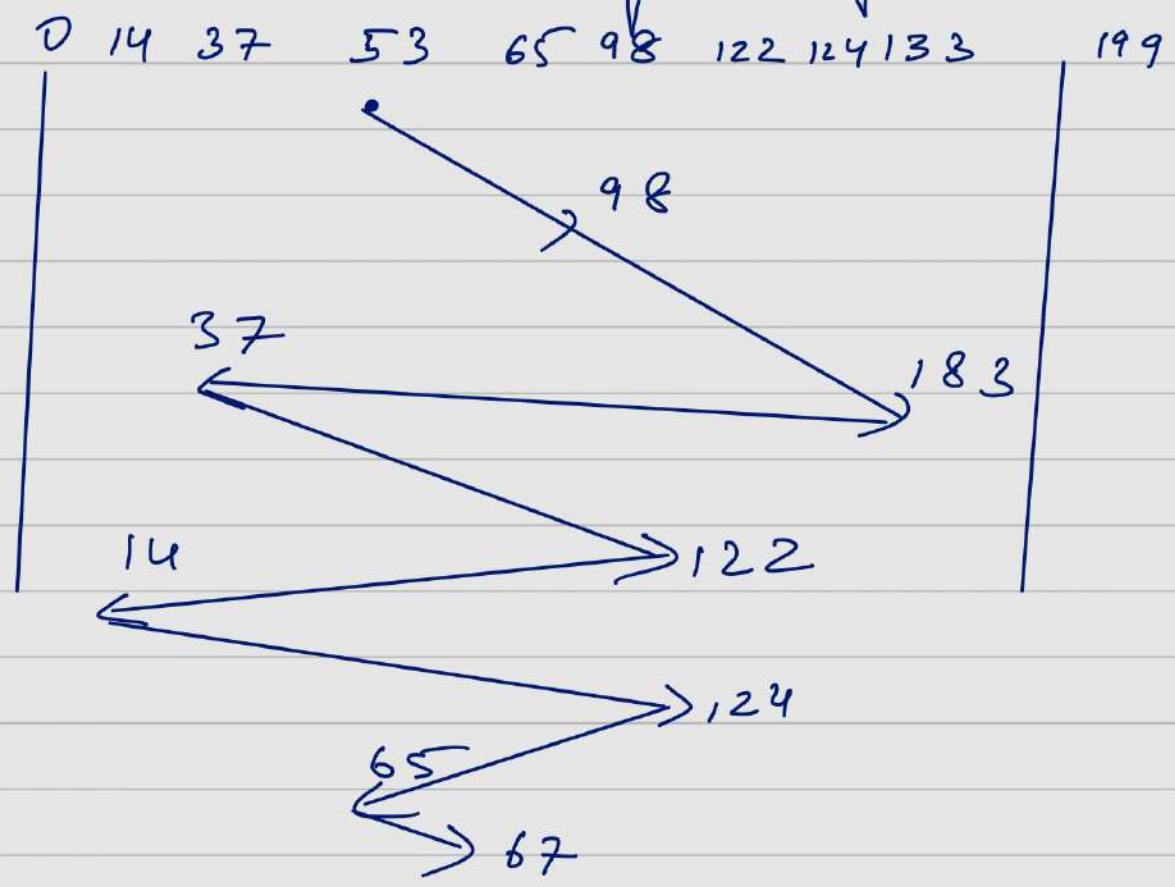
5 7 120 150

Minimize the movement of disk head.

(1) FCFS (First Come First Served)

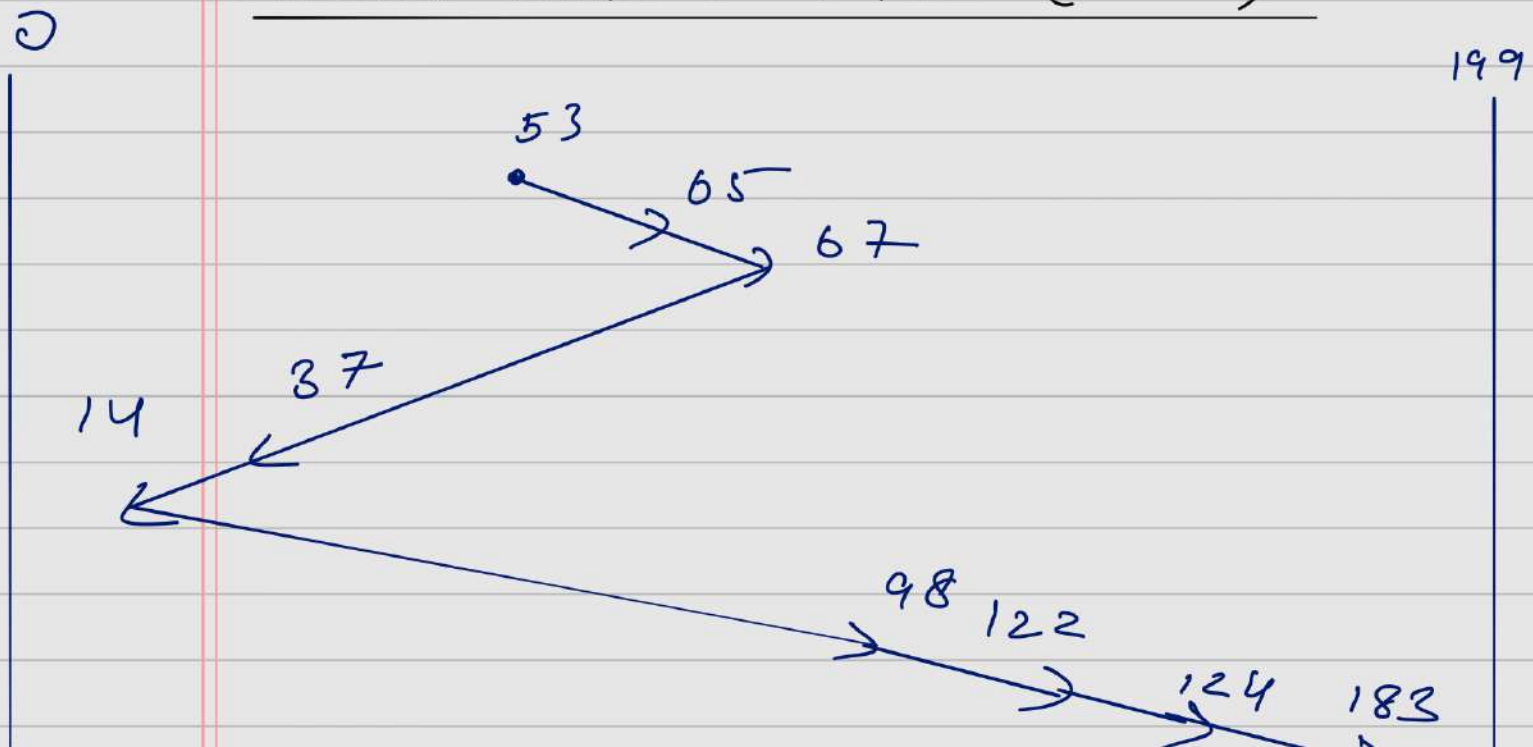
98, 183, 37, 122, 14, 124, 65, 67

Disk head is initially at cylinder 53.



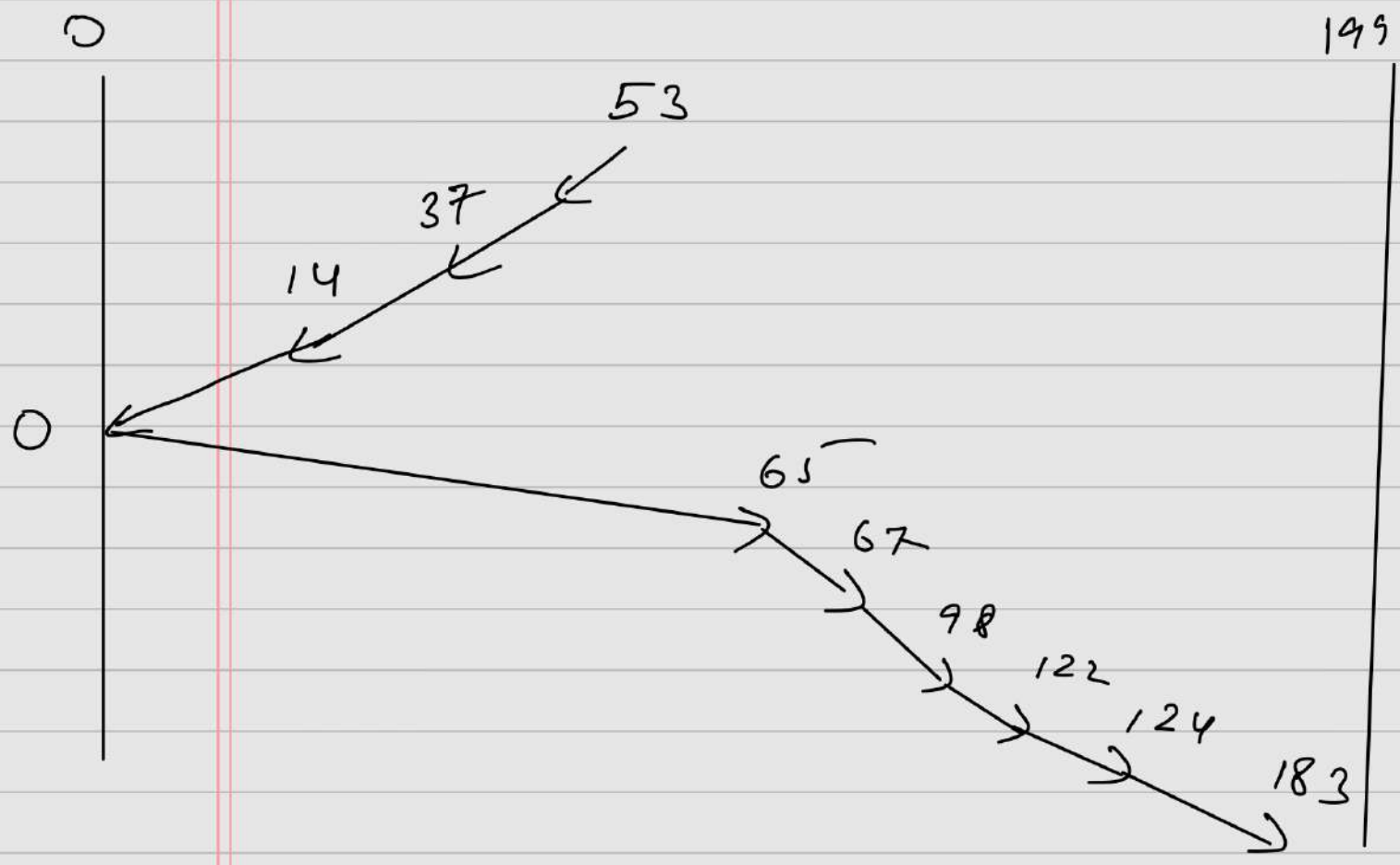
$$(98-53)+(183-98)+(183-37)$$
$$+(122-37)+(122-14)+(124-14)$$
$$+(124-65)+(67-65)=640$$

Shortest Seek Time First (SSTF)

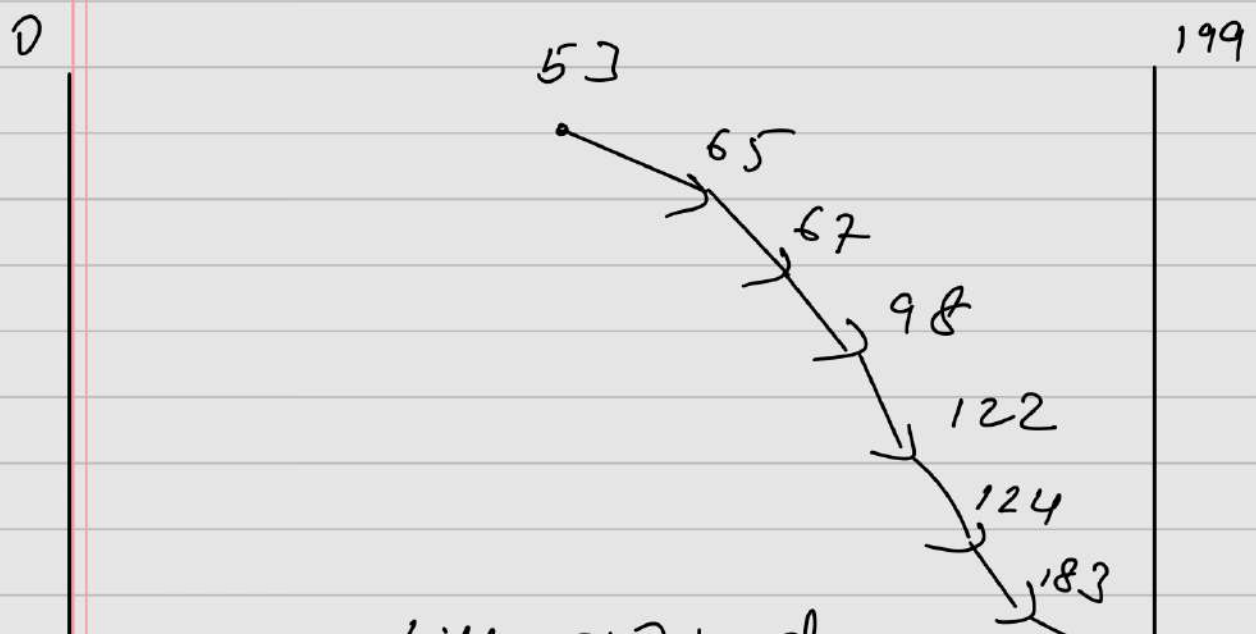


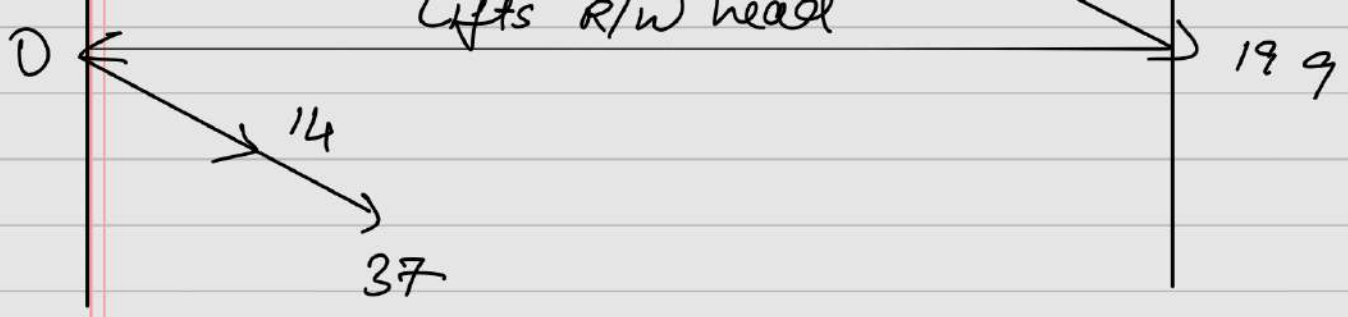
Movement → 236

SCAN → LOOK



C-Scan → C-LOOK





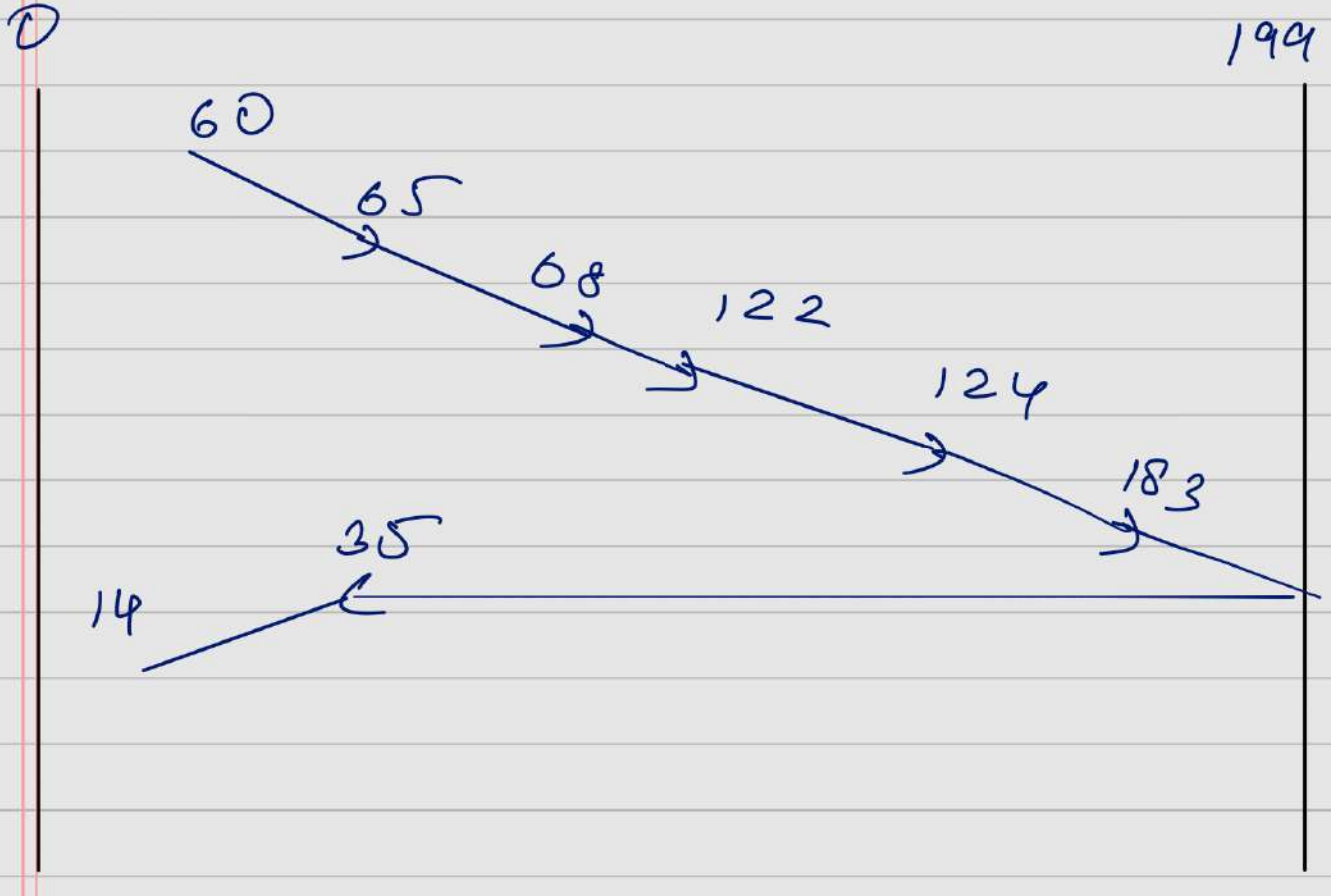
LOOK & C-LOOK have only difference in direction

16/4/25

SCAN

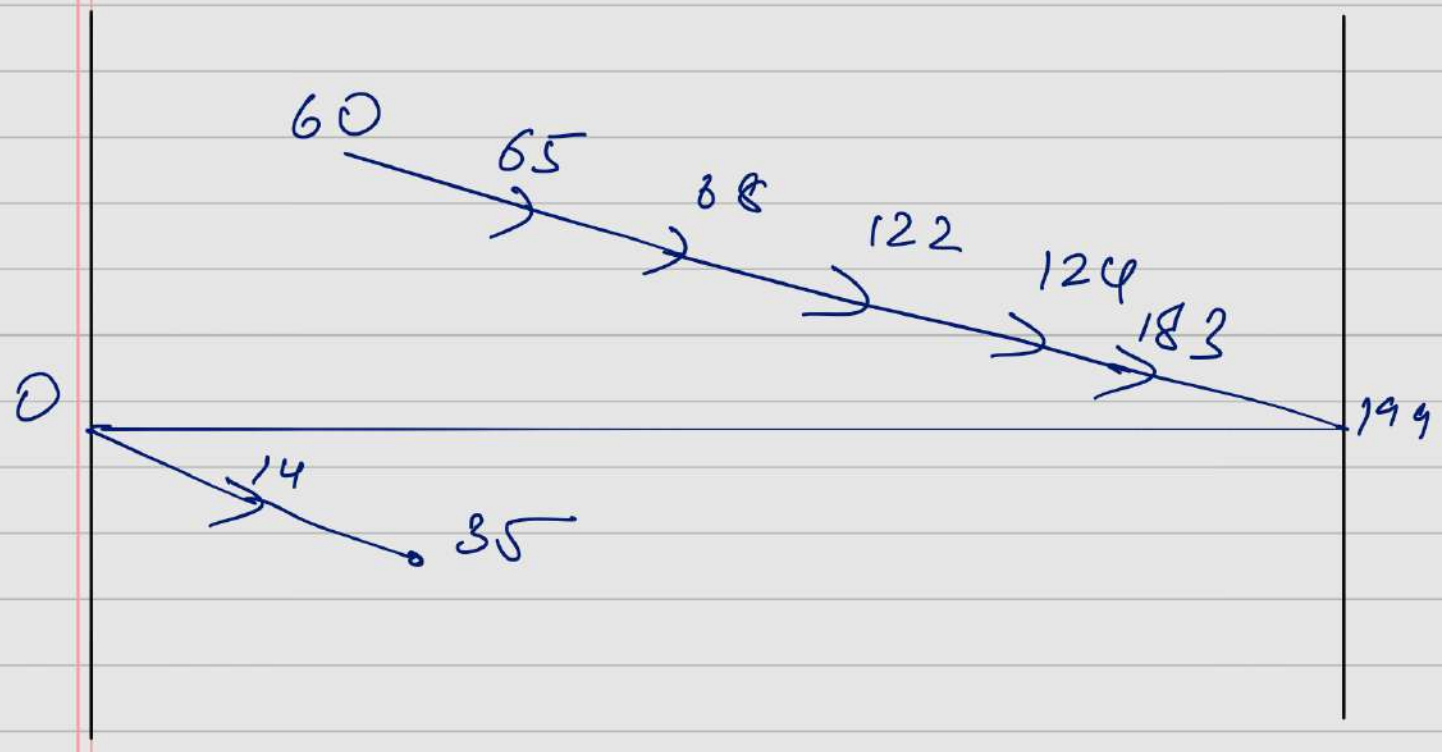
14 65 122 37 68 124 183

R/W head is at 60



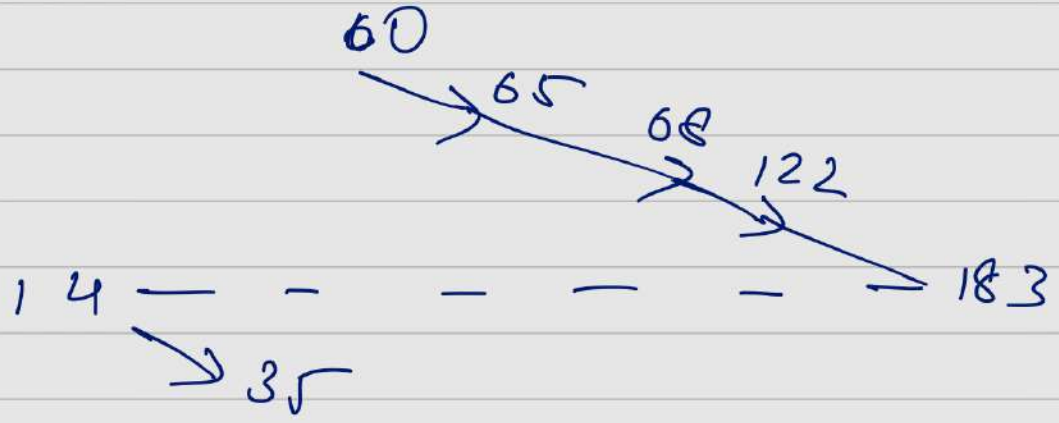
C-SCAN

0 199

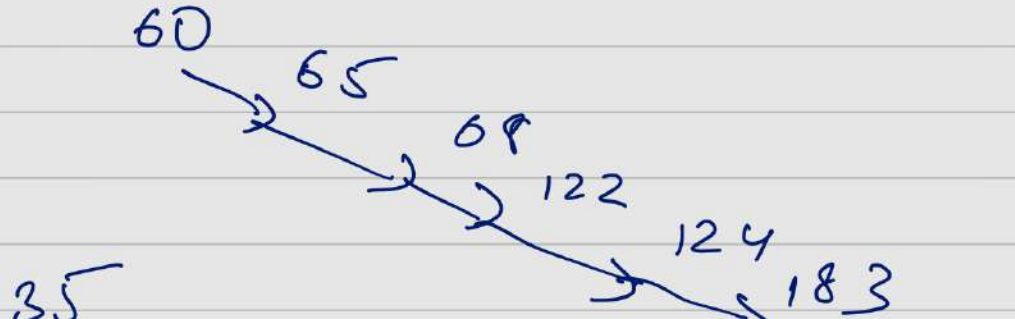


$$(65-60)+(68-65)+(122-68)+(183-122)$$
$$+(199-183)+(199-0)+(14-0)+(35-14)$$

C-LOOK



LOOK



File System

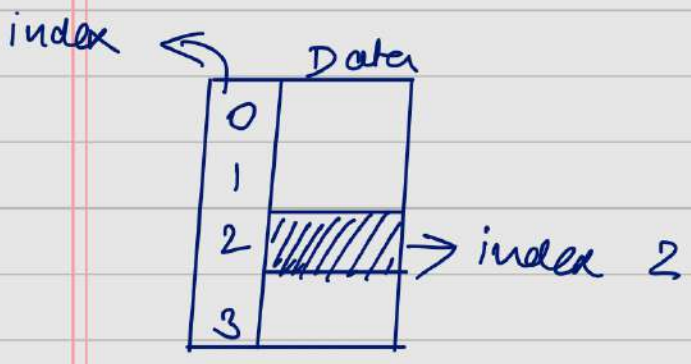
- text files
- source files → sequence of func^n/ declarations followed by some executable statement.
- executable files → series of code sections that the loader can bring into the memory for execution.

Access Methods

- Sequential Access (compiling programs, text editors)
- Direct Access - file is broken down into fixed-length logical records
  ↓
  read records in no particular order
- Index-based Access

(Other access)

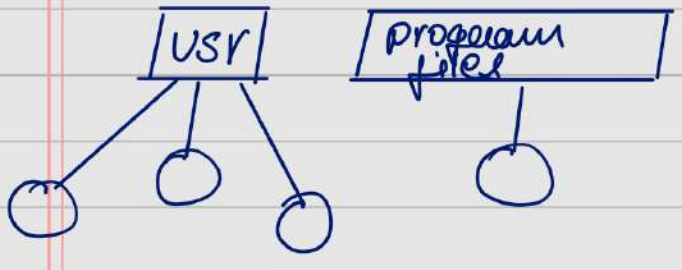
Similar to Paging



Directory Overview

- Single Level Directory

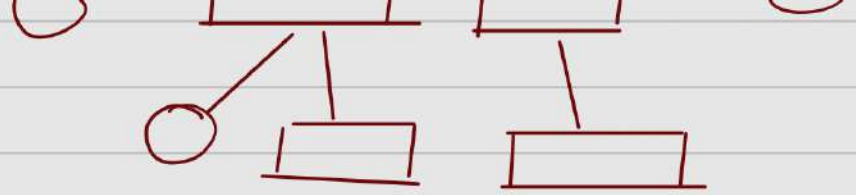
Cannot support multiple user



- Two Level Directory ⇒ Each user will have its own user file directory (UFD)

Master File Directory - keeps track of all user file directory (MFD)

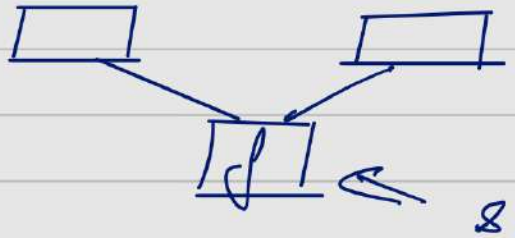
Tree Structured Directory



Absolute Path → path from the root of the directory

Relative Path → path from the current place in the directory.

Acyclic Directory → only this directory can support this



shared file