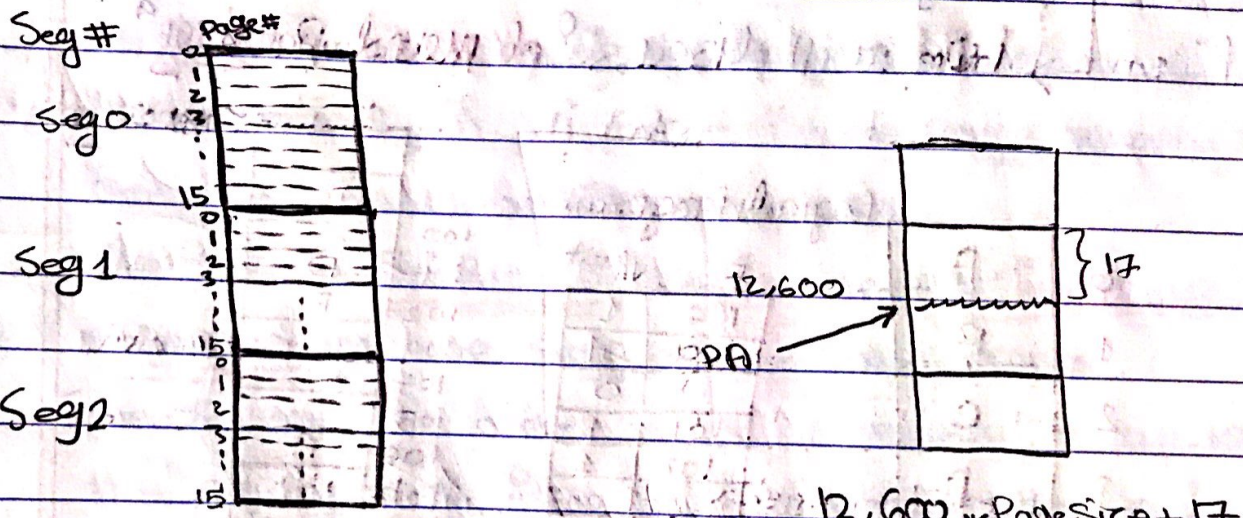
example: given LA which generates:

$S = 2, d = 350$

$PA = B + d = 18,000 + 350 = 18,350$

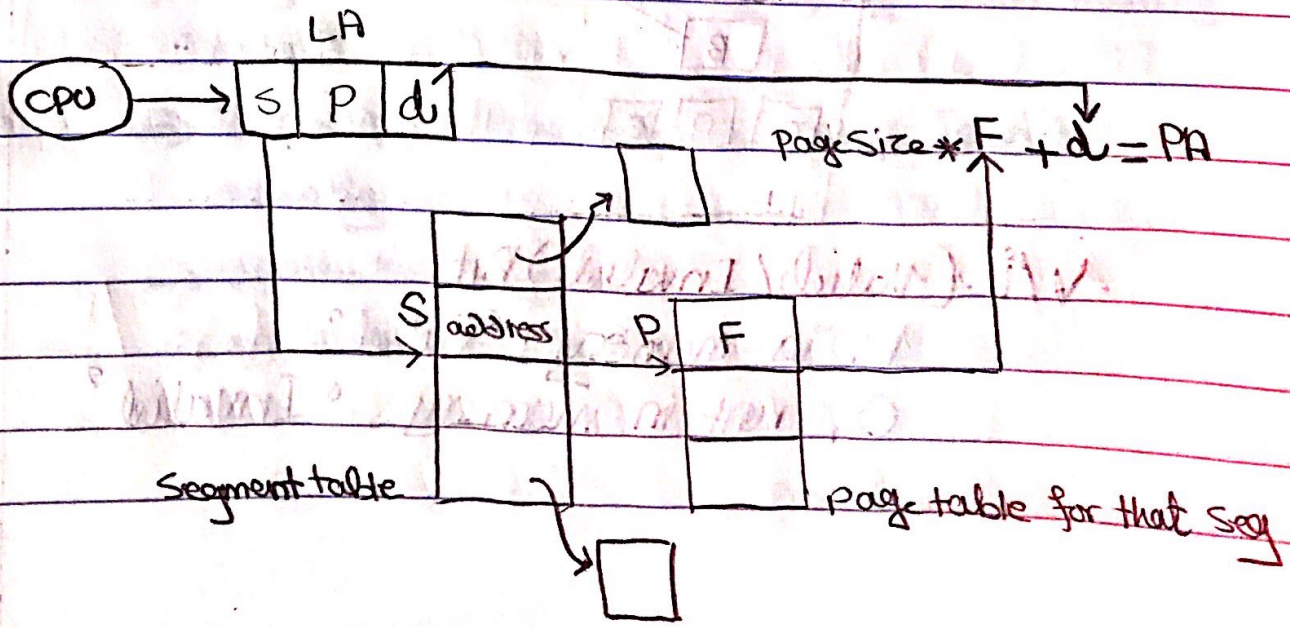
## Segmentation with Paging

"paging the Segments"




12,600 * PageSize + 17

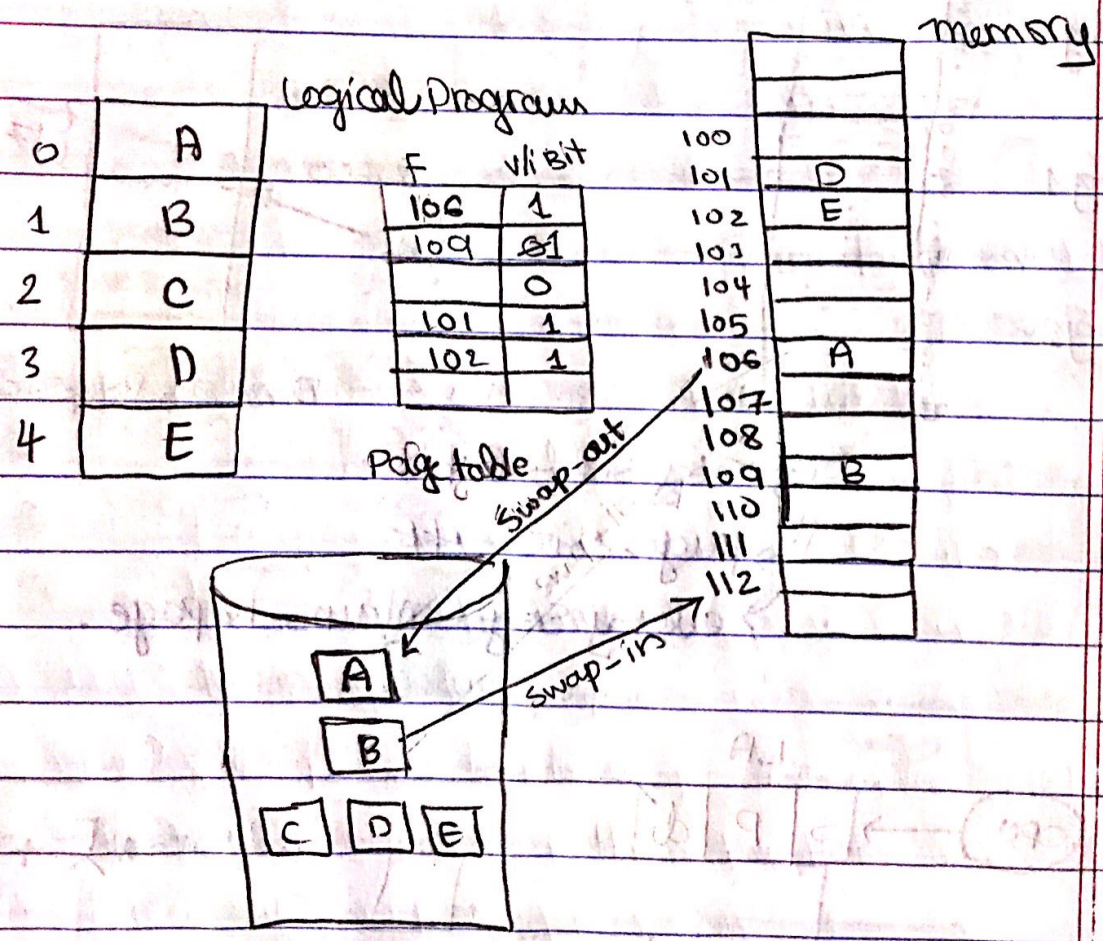
- If seg size = 64K.
  - → page size = 4K.
  - → every seg contains 16 page.



# Chapter #9:

## Virtual Memory Management

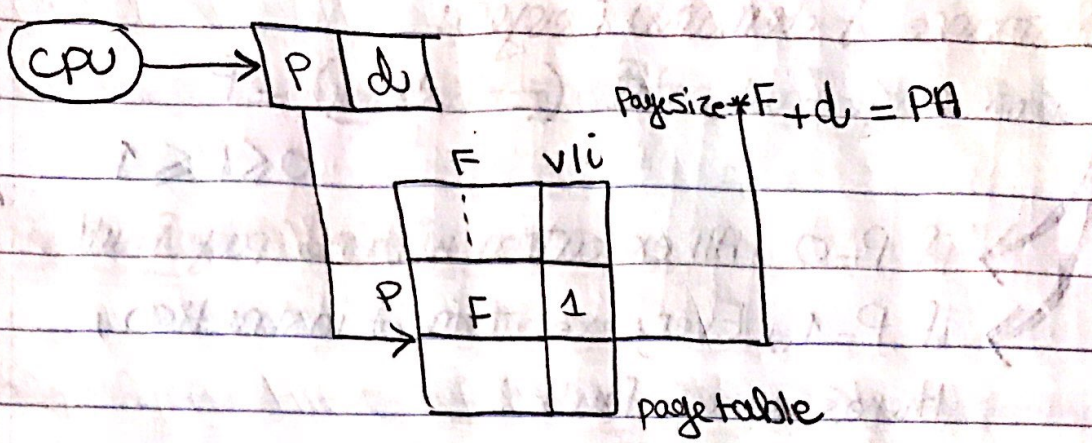
- How to run bigger programs on smaller memory.
- No need for the whole program to be Loaded in to memory, only part of it will do.
- We will discuss "demand Paging"




### v/i (valid/Invalid) Bit:

1; "in memory. "valid"

0; not in memory. "Invalid"




- The LA is checked through the page table if it's a valid address, that's, if V/i Bit = 1 (the page memory) that it continue execution as usual.
- What if the V/i Bit = 0, that's the page isn't in memory, then we say a page fault occurs.
- The OS looks for a free frame in memory, swap-in the required page from HD, it updates the page table & resumes execution.
- What if there's no free frame?
  - → It selects a victim frame
  - → may be, it swap-out the page from memory to HD.
  - → It swap in the required page from HD to memory
  - → update the page table.
  - → resumes execution.

## Performance of demand pages:

- Assum page fault rate (probability) = P

  $0 \leq P \leq 1$

  - If P=0, All execution with no page fault. "perfect"
  - If P=1, Every execution of instruction, there's a page fault.

- Assum memory access = m

EAT = (1-P) * m + P * [ Swap in Page + maybe swap-out page + m ]

Page fault overhead.

Example: Assum memory access (m) = 10 mics

- Page fault rate (P)
- Page transfer (swap) time = 10 mils
- 40% of the time the page needs to be swap-out. (swap-in)

EAT = (1-P) * 10 + P * [10 * 1000 + 0.4 * 10 * 1000 + 10]

= 10 - 10P + 10,000P + 4,000P + 10P

= 10 + 14,000P ≃ 14,000P

⚠ تحويل الوحدات الأكبر لوحدات أصغر

mils, mics → mils ⇒ mics

∴ Result:

Performance depends on PF (page Fault Rate).

Objective: Minimize the Page Fault Rate.

⚠ Note: A new bit called "dirty Bit" is added to the page table to indicate if the page is modified.

1 ≡ page is modified.

0 ≡ page isn't Modified.

Page Replacement Algorithms:

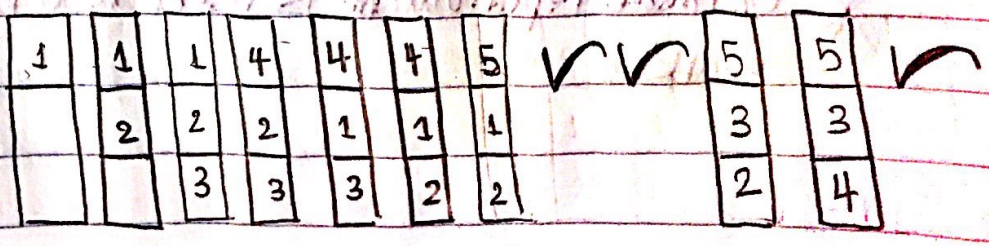
The OS must select the victim frame so that, to minimize the page fault rate.

## 1 FIFO

Replace the page which enters the memory First (oldest page in memory).

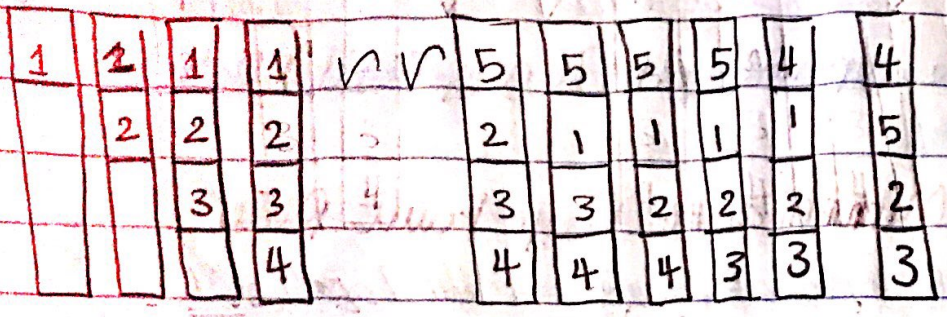
example: given the following page references.

1 2 3 4 1 2 5 1 2 3 4 5




9 page faults

1 2 3 4 1 2 5 1 2 3 4 5



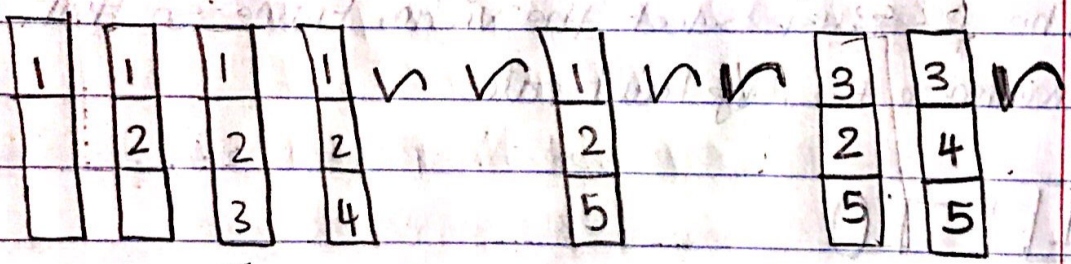
10 page faults

∴ Belady's Anomaly.

[2] Optimal Replacement:

Replace the page which won't be used in the future for the longest period.

1 2 3 4 1 2 5 1 2 3 4 5



7 Page faults

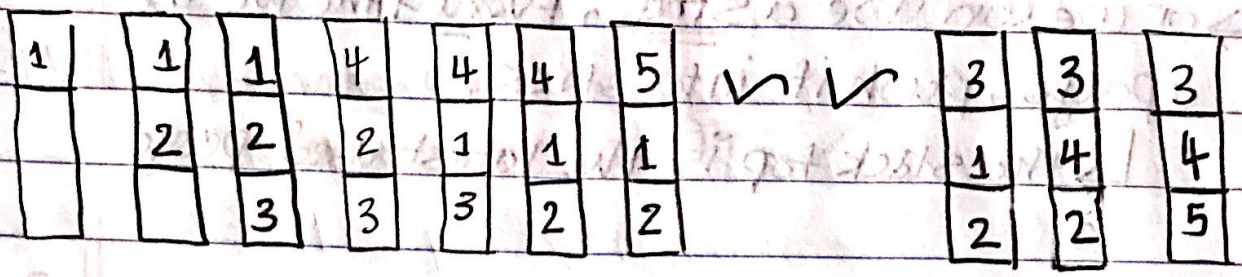
[!] Major problem: How the OS know in advance the next page in which execution occurs?!

* Optimal replacement is used as a bench mark.

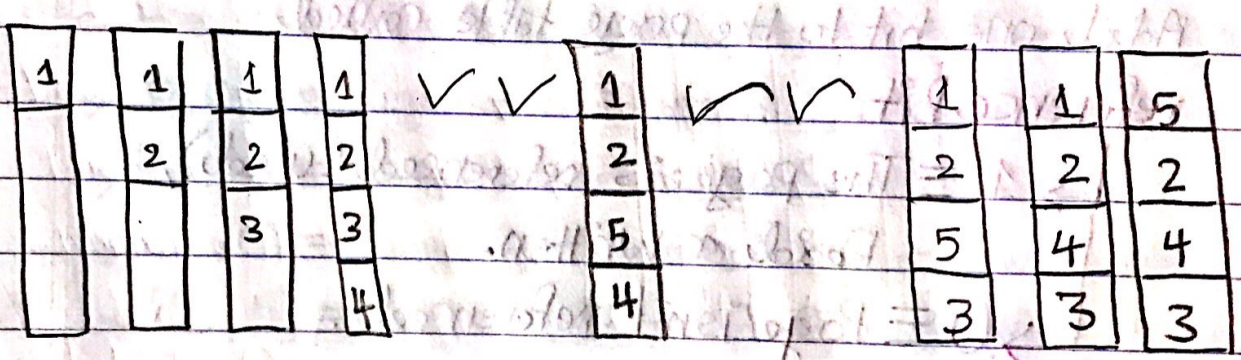
## 3 Least Recently Used [LRU]

Replace the page which have not been used in the past for the Longest period of time.

1 2 3 4 1 2 5 1 2 3 4 5




10 page faults "3 Frames"




8 page faults "4 frames"

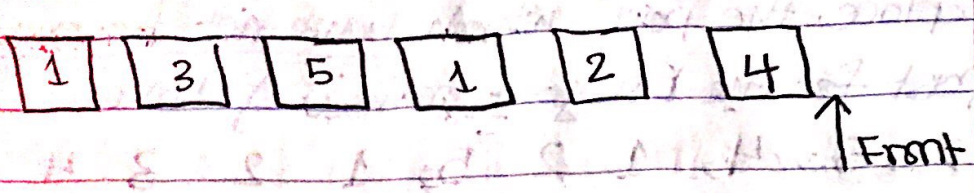
### * Implementation:

(1) Add a Counter to the page table that contains the last time the page is referenced (used)

| | time of reference |
|---|---|
| | 10:3:20 → Integer |
| | 10:3:11 |
| | |

page table

(2) Keep a queue of referenced pages

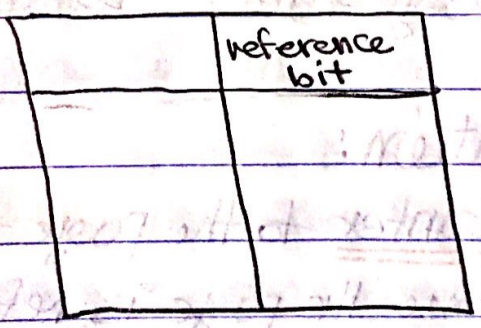



→ or we can use a stack, Every time we use a page, push it into stack.
↳ the stack top is the latest used page

## LRU approximation:

Add one bit to the page table called reference bit

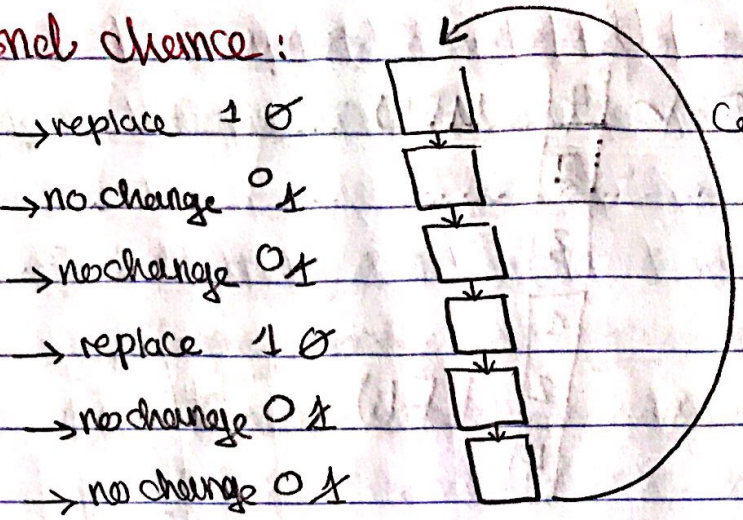
- → 1 ≡ The page is referenced (used), Read or Written.
- → 0 ≡ Page isn't referenced.




Page table.

→ LRU approximation, Replace the page whose reference Bit is 0.

## Second chance:




## Enhanced Second chance:

Use the reference bit & the dirty bit. (Reference bit, dirty bit)

best
- → (0,0) ≡ Page isn't referenced & not modified. (1)
- → (0,1) ≡ Page isn't referenced but modified. (3)
- → (1,0) ≡ Page is referenced but not modified. (2)

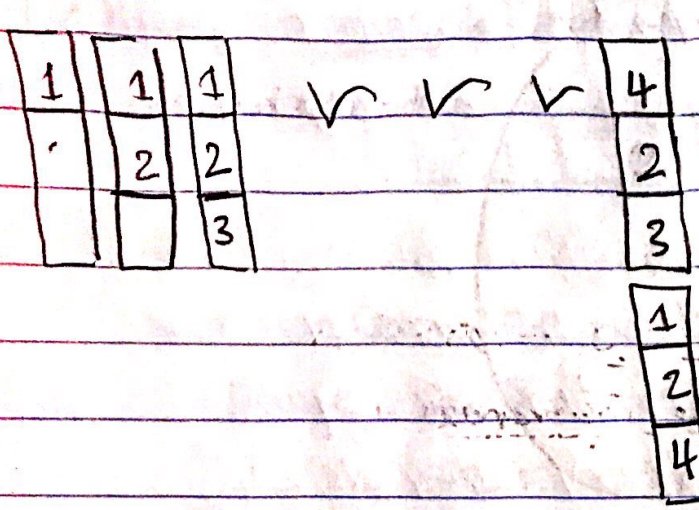
worst
- → (1,1) ≡ Page is referenced & modified. (4)

## ✱ Counting Algorithms:

1_ MFU (Most Frequently Used)
Replace the page which has been used for the maximum # of all times.

2_ LFU (Least Frequently Used)
Replace the page which has been used for the minimum # of all times.

1 2 3 1 2 1 4 3 4 5




## Global vs. Local Replacement

May. 19. 2018

## Allocation of Frames for processes :-

(1) Equal Allocation : Each process gets the same number of pages (frames).

example: memory 100 frames, 5 processes

- Every process gets $\frac{100}{5} = 20$ frames.
- unfair → not good performance.

(2) Proportional Allocation according to Size :

example: Assume the size of process $P_i = S_i$

Assume we have m frames of memory

Process $P_i$ gets $\frac{S_i}{\sum S_i} * m$

- Assum memory 100 frames, 3 processes with sizes 100, 400, 700 KB.
- $P_1 = \frac{100}{1200} * 100 \simeq 8$ frames
- $P_2 = \frac{400}{1200} * 100 \simeq 34$ frames
- $P_3 = \frac{700}{1200} * 100 \simeq 58$ frames.

(3) Proportional according to priority:

example:

| process | priority |
|---|---|
| $P_1$ | 2 |
| $P_2$ | 3 |
| $P_3$ | 7 → highest priority |

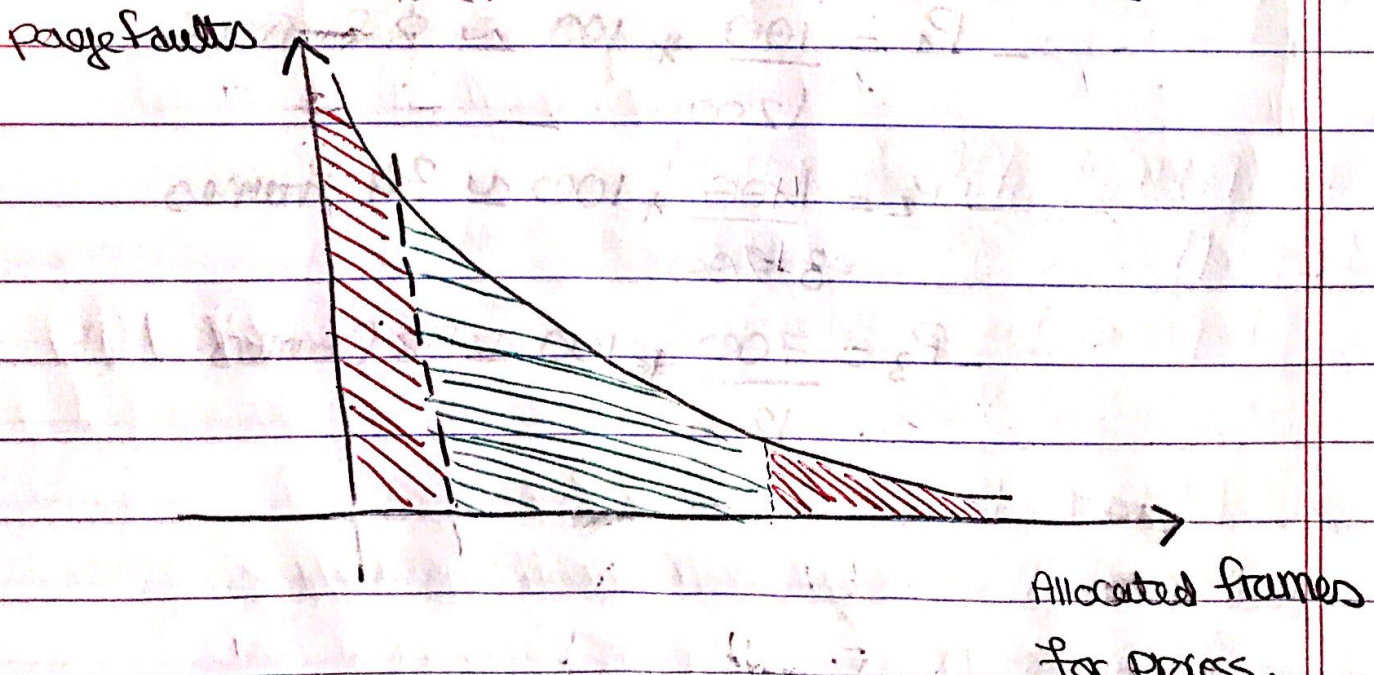
- If we have 100 frames of memory

- $P_1 = \frac{2}{12} * 100 = \frac{1}{6} * 100$

- $P_2 = \frac{3}{12} * 100 = \frac{1}{4} * 100 = 25$ frames

- $P_3 = \frac{7}{12} * 100$

Thrashing: The OS is busy swapping page in & out.




Good performance.

Poor performance.

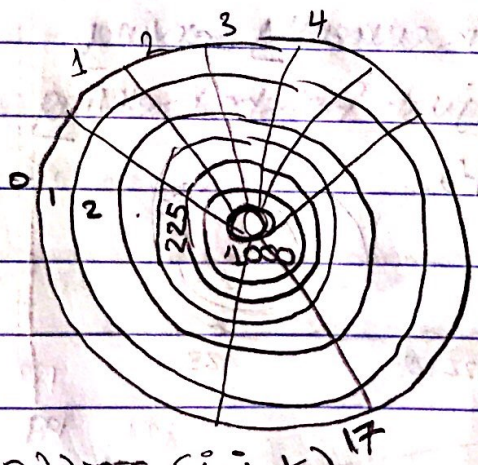
- Sometimes the number of Allocated frames for the process is low, which leads to poor performance (low System utilization).

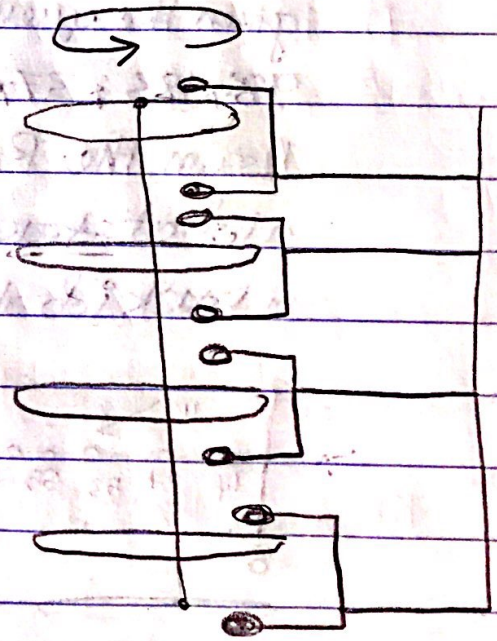
The OS thinks that the degree of multiprogramming is low,

→ OS increases the degree of multi programming.

→ The system gets worse.

→ Most of the time the system OS is doing swapping of page.

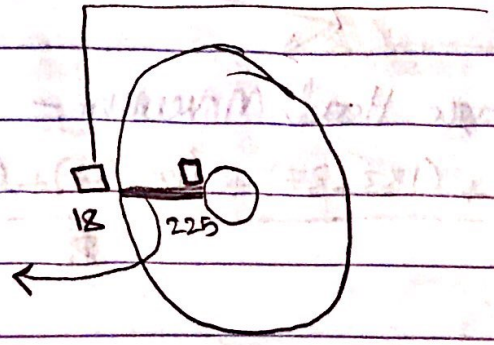


Address (i, j, k)

track # ← surface # → block #

(225, 3, 17)



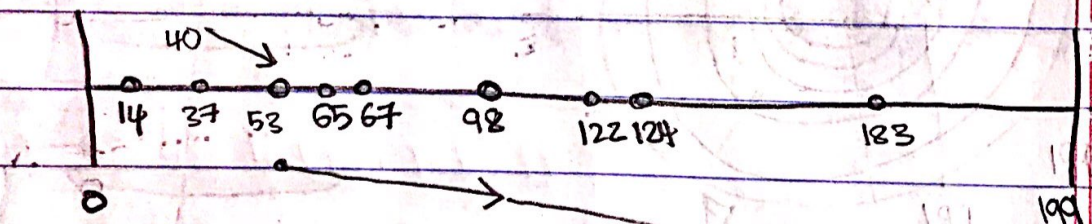

The time to move from track 18 → 225 is called "seek time" mechanical move.

HD access time = Seek time + Latency time + transfer time

We have no control

→ we could improve the seek time by selecting a good disk scheduling algorithm. (minimize seek time)

example: Assum HD has 200 tracks (0 — 199)
given the queue of HD requests as follows:
98, 183, 37, 122, 14, 124, 65, 67.
Assum the R/W head is currently serving a job at track 53 & just finished serving a track job at track 40.

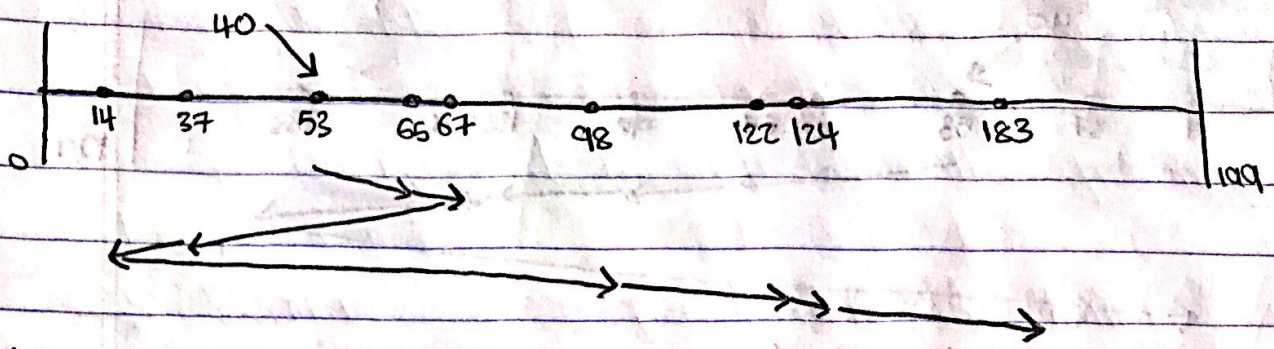



(1) FCFS

→ Average Head Movement =

$$\frac{(183-53)+(183-37)+(122-37)+(122-14)+(124-65)+(67-65)}{8}$$

## (2) Shortest SeekTime First (SSTF)

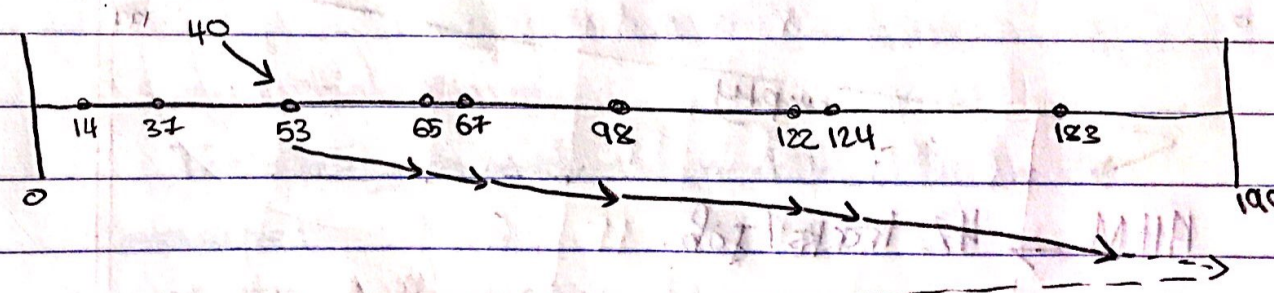



AHM → 29 tracks / Job.

SSTF gives the optimum solution
minimum

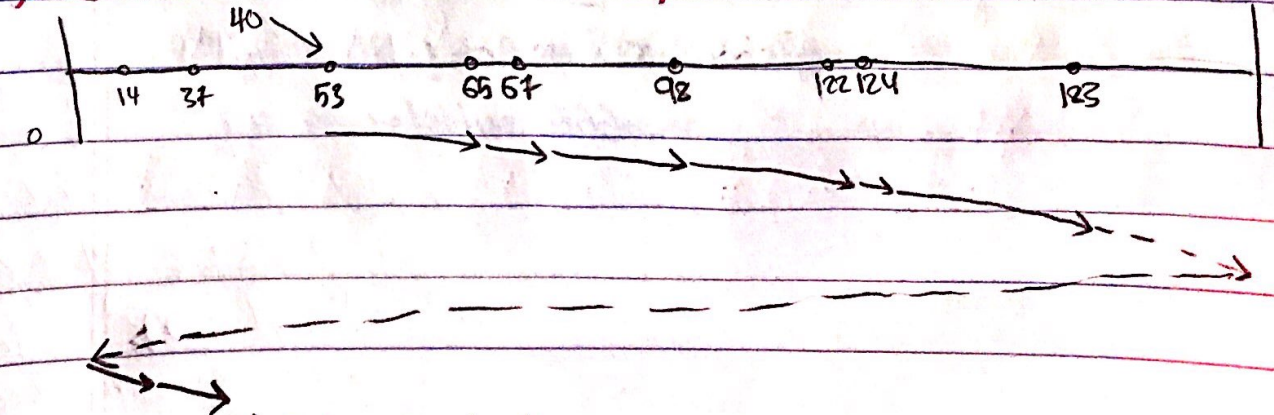
⚠ Problem: Starvation

## (3) SCAN (Elevator)



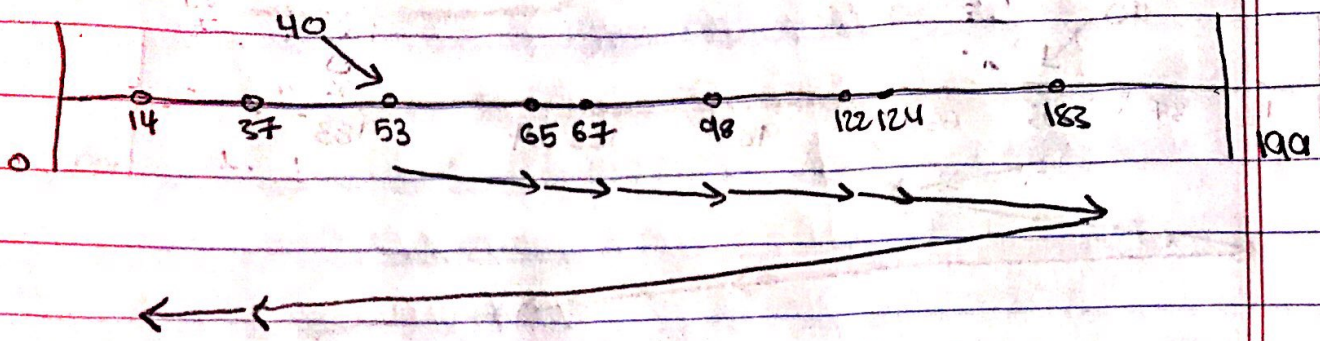

AHM → 41 tracks / Job.

## (4) C-SCAN (Circular SCAN)



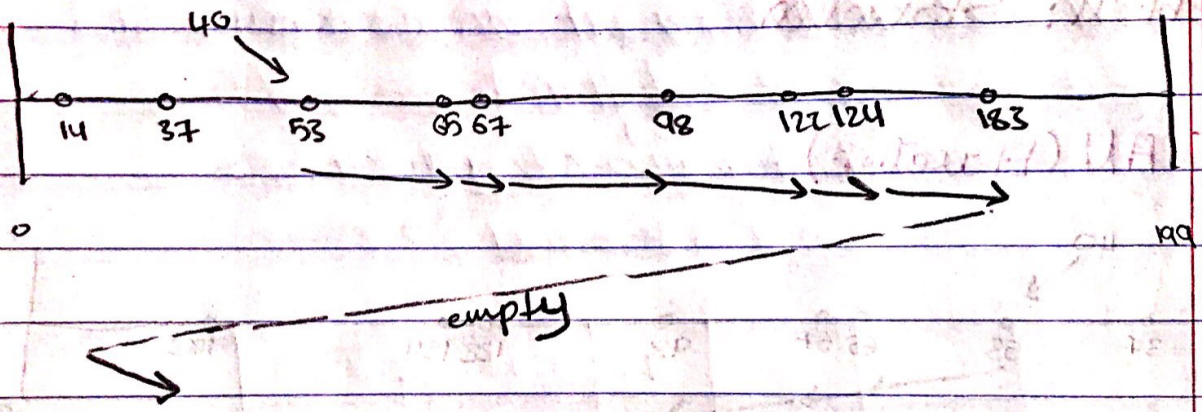

AHM → 47 tracks / Job.

## (5) Look:

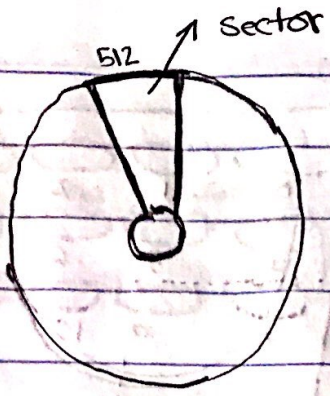


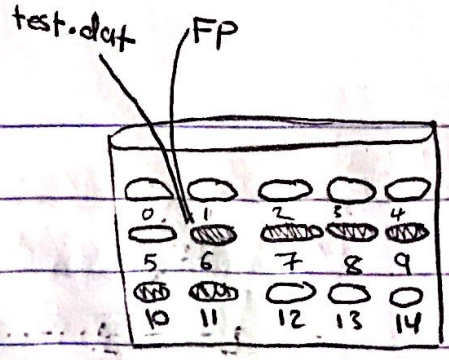

AHM → 37 tracks/Job.

## (7) C-Look:




AHM → 42 tracks/Job.




Hard Disk

# File Access:

(1) Sequential Access.

(2) Direct Access.

- Location: Address of First block in the file (constant, can't be changed).
- File Pointer: The current location where the file is located.

→ Sequential Access:

The ability to read the next block in the file.

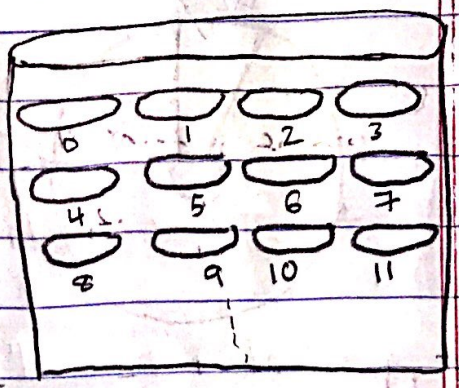
example:

If FP at block (7), then sequential access is the ability to read/write block (8).

→ Direct Access:

The ability to read/write block # in the file

(n is relative Address 'في الملف' Block)




## Allocation Methods:

How the disk blocks are selected and allocated for the file.

**Note:** Every storage device, has "device directory" which contains the information about the files stored on that device.
Generally, it's a "hash table".

Device directory

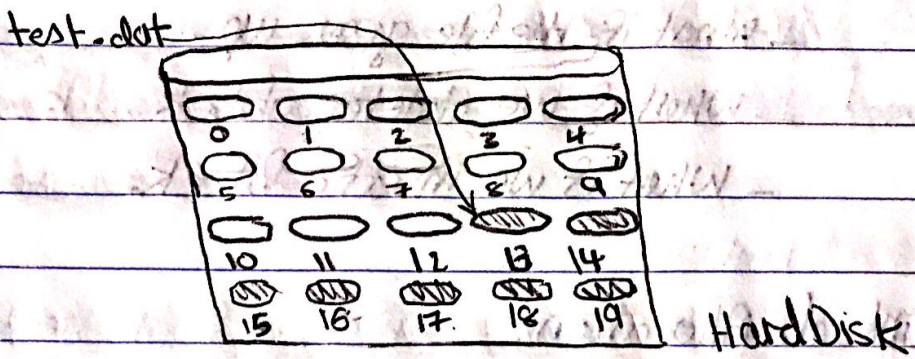
| Name | Location | FP | Size | |
|---|---|---|---|---|
| sam.txt | 11 | 11 | 6 | |
| All files on the device | | | | |

## (1) Contigious Allocation:

The OS selects contigious blocks of the file.

## Device Directory

| Name | Location | Size | FP |
|---|---|---|---|
| test.dat | 13 | 7 | 13 |
| | | | |




∴ Advantages:

It supports both sequential & direct access easily.

→ Sequential Access:

If FP at address X,

∴ The next block address is X+1.

→ Direct Access:

Assum we want to read block #5 (5 relative) in the file

- The physical address of block #5 in the file =

Location + 5 − 1 = 17

∴ To access Block #n,

PA = Location + n − 1

## ⚠ Disadvantages:

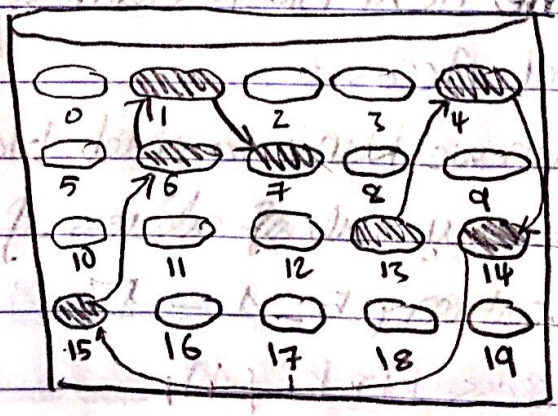
(1) Fragmentation (holes).
↳ Solution: Compaction (defrag).

(2) "Major Disadvantage"
- What if the file grows up.
- What if we delete a block.
- What if we insert a block.

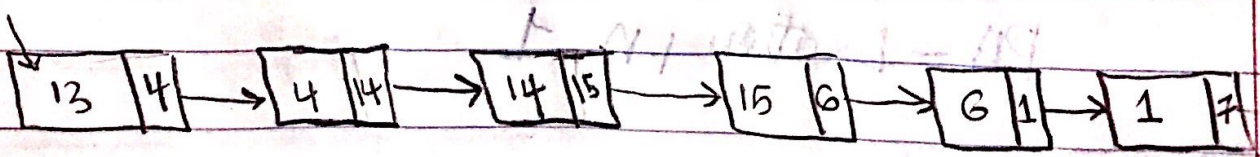
## (2) Linked Allocation:

- The file blocks are selected as a linked list of blocks.

| Name | Location | Size | |
|---|---|---|---|
| test.dat | 13 | 7 | |




location



∴ Advantages:

No fragmentation

- File can grow, insert, delete as you like.

⚠ Disadvantage: "Major"

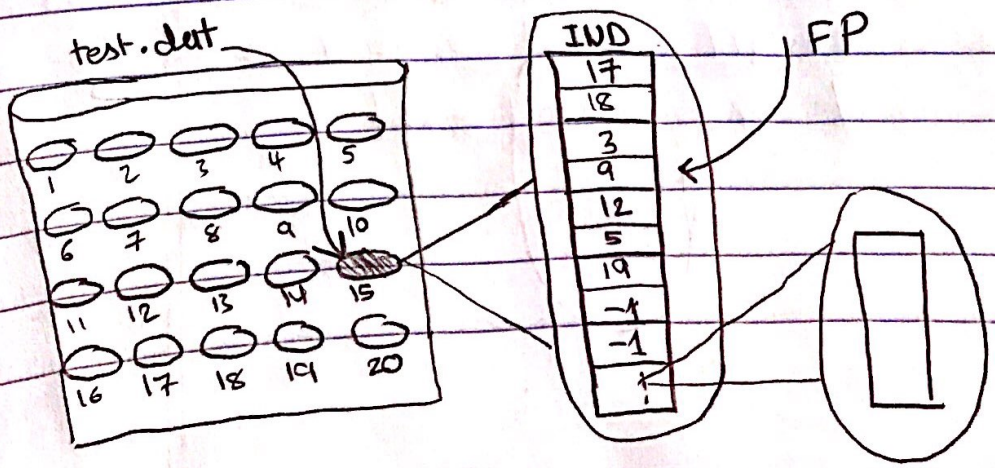
- It only supports Sequential Access easily.
- Direct Access isn't supported & needs other tools such as an "index file".

| 13 |
|---|
| 4 |
| 14 |
| 15 |
| 6 |
| 1 |
| 7 |

test_index.dat

(3) Indexed Allocation:

- Each file has at least one index block which contains the addresses of the file data blocks.



∴ Advantages:

It supports both Sequential & Direct Access.

→ Sequential Access:

IF FP at IND[i];

next block address equals IND[n++];

→ Direct Access:

To access block # n in the file, its PA = IND[n].

⚠ Disadvantage:

The wasted index Block(s). each file needs at least one index Block.