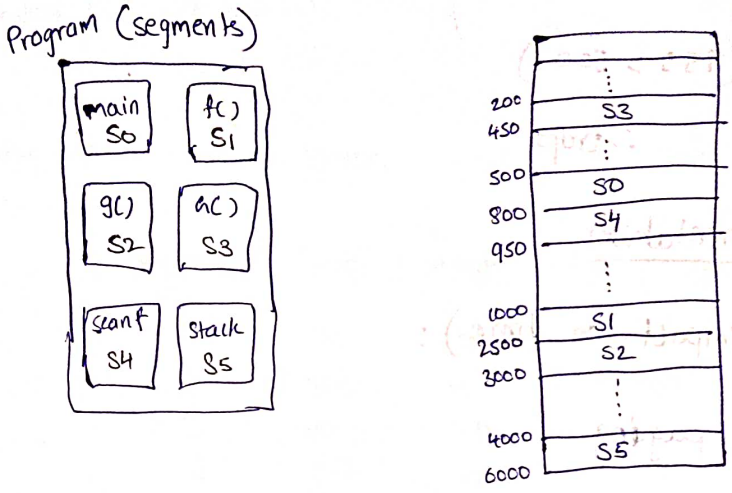
INDEX

NAME: K. Gowtham STD.: SEC.: ROLL NO.: SUB.:

II. Segmentation

Paging does not preserve users view of program



→ At user view program is divided into segments (each segment may be of diff sizes).

→ Each segment has segment id (assigned by loader)

→ Different segments a stored non-contiguously. However a segment itself is stored totally at one location (contiguously)

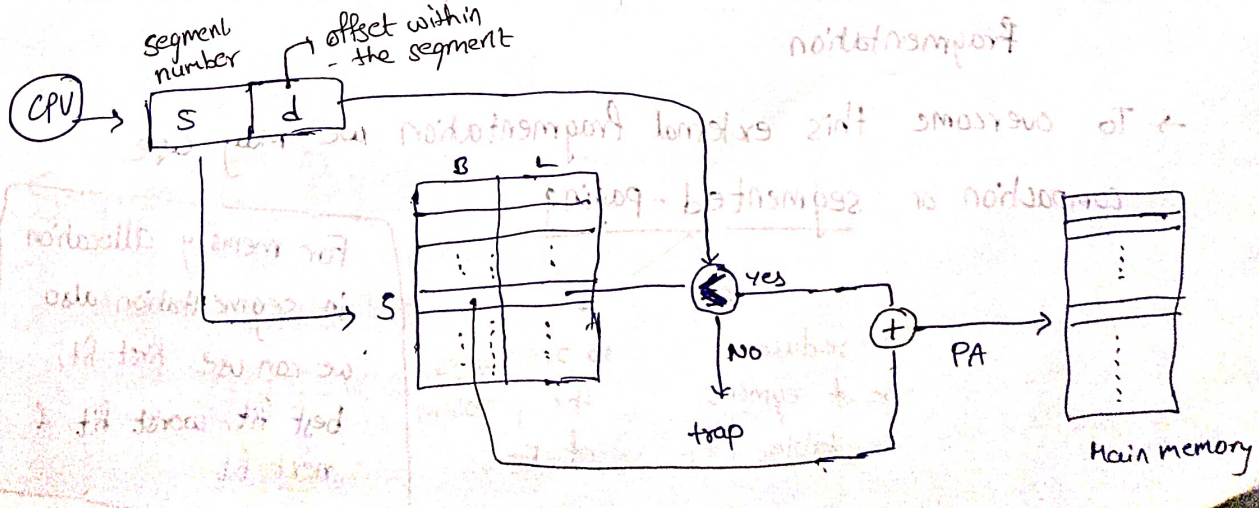
→ Here MMU is segment table.

	B	L
0	500	300
1	1000	1500
2	2500	500
3	200	250
4	800	150
5	4000	2000

B - Base address ~~of segment~~

L - length of segment

→ Every process has its own segment table



Eg: For above example

$LA\langle 3,50\rangle \longrightarrow PA\langle 200+50\rangle = 250$

Eg: $LA\langle 2,755\rangle$

(755 > 500)

∴ trap

Performance of Segmentation

(i) Temporal issue (impact on time):

* Exactly same as paging

EMAT = 2m without TLB

Silly we can add TLB & PAC (same equation as in the case of paging)

(ii) Spatial issue

* Large segment table is undesirable

* One of ways to reduce size of segment table is to use paging on segment table.

★ Note:

* In case of segmentation, we don't divide PAS.

∴ PAS is organized with variable partitioning (MVT)

we know that variable partitioning suffers from external fragmentation

→ To overcome this external fragmentation we may use compaction or segmented-paging

- reduces size of segment table
- to overcomes the problem of EF

For memory allocation in segmentation also we can use first fit, best fit, worst fit & next fit

Note :

	IF	EF
Paging	✓	✗
Segmentation	✗	✓

IF - Internal Fragmentation

EF - External Fragmentation

Q31) Consider below segment table

segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

what the physical addresses for following logical addresses

a) 0,430

219 + (430 - 0) = 219 + 430 = 649

b) 1,10

~~2300 + 9 = 2309~~ 2300 + 10 = 2310

c) 2,500

500 > 100

∴ trap

d) 3,400

1327 + 400 = 1727

e) 4,112

112 > 96

∴ trap

Segmented - Paging :

Assume LA = 34 bits

<s,d> = <18, 16> bits

⇒ Maximum segment size = 2^{16} = 64KB

total no of segments possible = 2^{18} = 256K segments.

→ To avoid EF we apply paging on segment.

(i) Divide address space (segment) into pages

(ii) Store these pages in frames.

(iii) Access the frames through page table of a segment

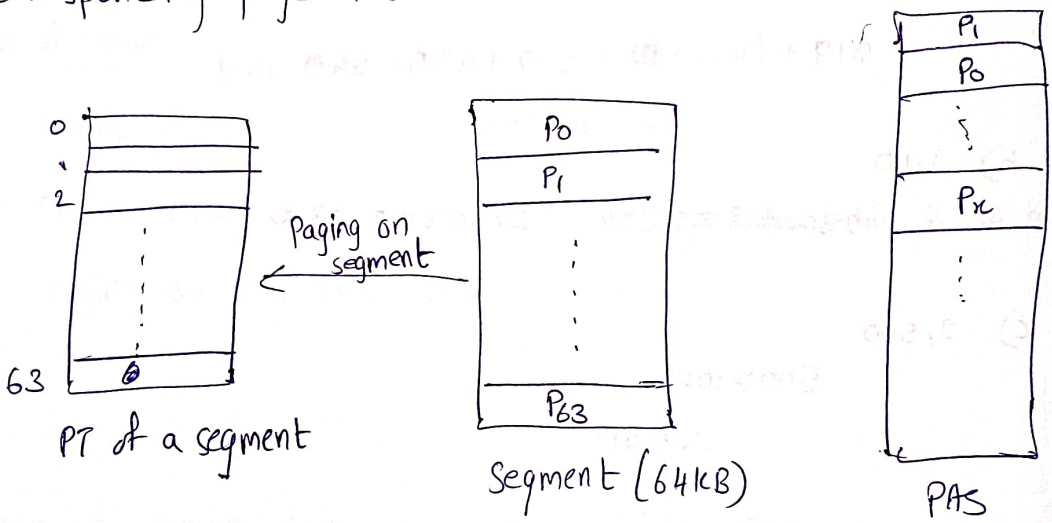
Let page size = 1 KB

no of pages in a segment = $\frac{64\ KB}{1KB}$ = 64 pages

→ Now PAS contains frames but not segments.

→ Now the page table of segment contains an entry for each page of the segment

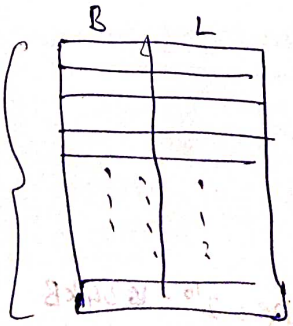
→ Now every segment must be accessed through its corresponding page table.



no of Page tables = no of segments.

Thus we have 256k page tables.

Structure of segment table:

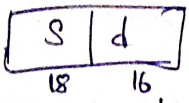


256k entries
(i.e., no of segments
= no of page tables)

B - base address of corresponding page table of the segment

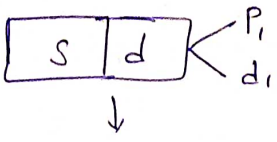
L - length of segment

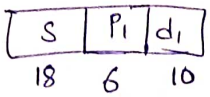
Logical address format



we have applied paging on segment
i.e., on 'd'

∴



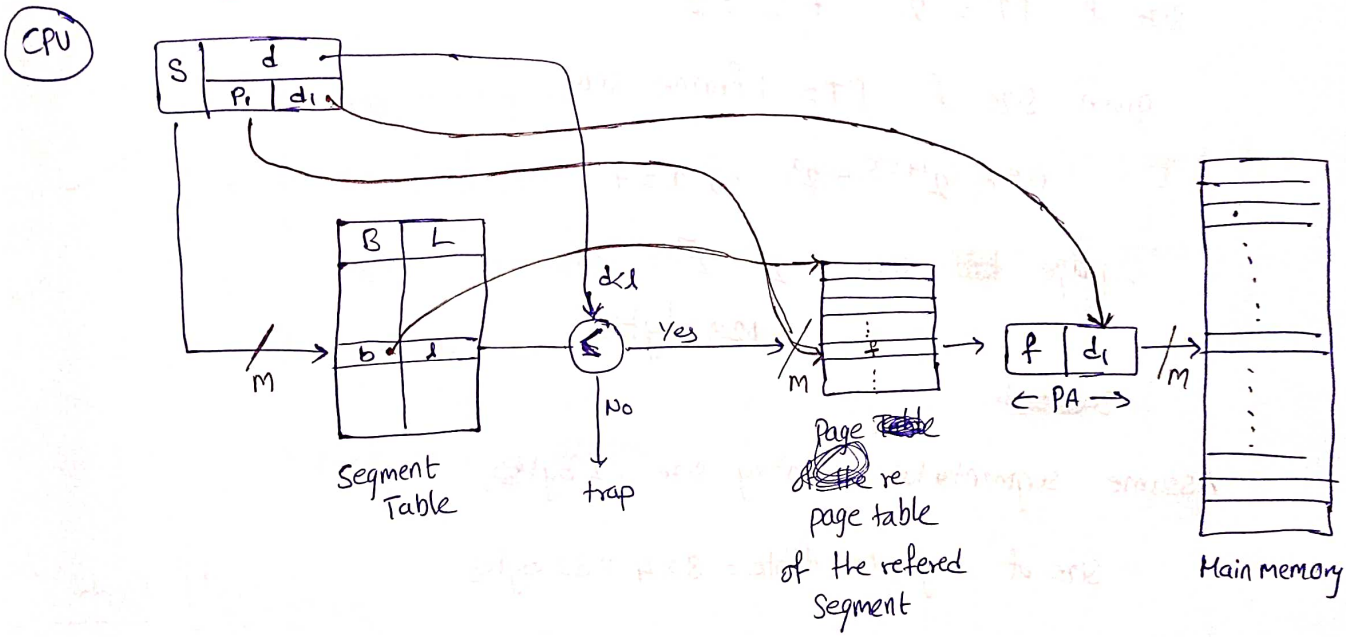


2^{P_1} - no of entries in each page table

2^{d_1} - size of page

d_1 say offset within the page

Address Translation



* Here we need to access segment table, then corresponding page table then the required data.

i.e., 3 memory accesses

∴ EMAT = 3M

Like in the case of paging we can use TLB to reduce EMAT.

Here also the TLB contains VA & corresponding PA.

(H4/14) VAS = PAS = 2^{16} bytes = 64 kB

no of segments in VAS = 8

size of each segment = $\frac{2^{16}}{2^3} = 2^{13}$ = 8 kB

VA:

s	d
3	13

Pg table entry size = 2 bytes

Let page size = 2^x

no of pages in a segment = $\frac{2^{13}}{2^x} = 2^{13-x}$

i.e., no of entries in a PT = 2^{13-x}

Size of PT = $2^{13-x} * 2 = 2^{14-x}$

given Size of PT = 1 frame size

i.e., $2^{14-x} = 2^x \Rightarrow x = 7$

∴ page size = $2^x = 2^7$

= 128 bytes

Assume segment table entry size = 4 bytes

Size of segment table = 8 × 4 = 32 bytes

(H4/15) 256 entry page table

=> no of pages in segment = 256 = 2^8

page size = 8kB = 2^{13} bytes

=> size of segment = $2^8 \times 2^{13} = 2^{21}$ bytes

VAS = 2k segments

= $2^{20} \times 2^{21} = 2^{41}$ bytes

given PTES = 2 bytes ; segment table entry size = 4 bytes (STES)

a) VAS = 2^{41} bytes = 2TB

b) Address translation space overhead in bytes:

here the overhead is size of seg. table & size of page table

size of segment table = $2^{20} \times 4 = 2^{22}$ bytes

size of each page table = $256 \times 2 = 2^9$ bytes

no of page tables = no of segments = 2^{20}

∴ overhead of page tables = $2^{20} \times 2^9$

$= 2^{29}$ bytes

∴ total overhead = $2^{29} + 2^{22}$ = 512 MB + 4MB

= 516 MB

c) for address translation

(i) access segment table for required segment's page table

(ii) access the page table for required frame of memory

(iii) accessing required word from memory

∴ 3 level memory access

Virtual Memory (VM):

VM gives an illusion to the programmer that a huge amount of memory is available for executing larger programs in small memory area.

→ It is a technique that supports the goal of managing execution of larger programs in small memory area.

→ CPU assumes that it can execute a program as big as VAS.

→ However PAS may not be as big as VAS.
So we need to store the program on disk.
From disk we load required pages into memory.

→ Sometimes required page may not be found in main memory. In that case required page is loaded into memory from the disk. This is known demand paging.

Demand paging:

Loading page from disk to memory at runtime based on demand is known as demand paging.

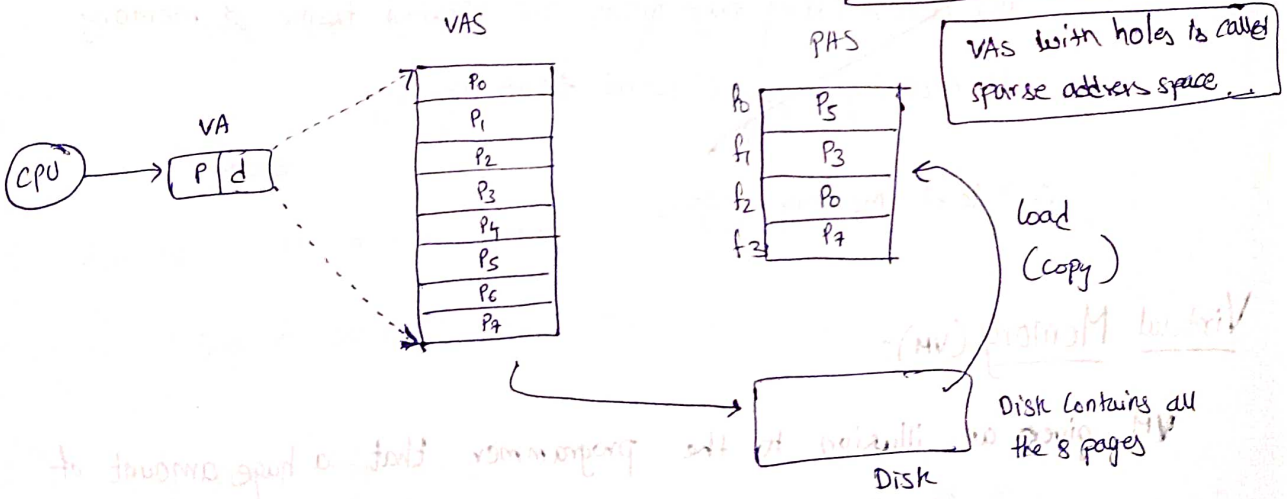
i.e., Virtual memory is implemented through demand paging.

VM implementation through Demand Paging:

Let VAS = 8KB ; PAS = 4KB ; PS = 1KB

⇒ N = 8 M = 4 d = 10

Sharing:
→ System libraries are shared by mapping system libraries into VAs
→ Silly processes can share mem for process communication.
Shared memory region is considered to be part of both the processes' VAS



→ Whether the req page is present in the memory or not is known by valid/invalid (V/I) bit which is in the page table.

→ If required page is found in the memory then it is called page hit.

→ If required page is not found in the memory then it is called page miss or page fault.

0	10	1
1	–	0
2	–	0
3	01	1
4	00	0
5	00	1
6	–	0
7	11	1

Page table

Page fault service:

1. Process that caused page fault gets blocked
2. Virtual memory manager now starts running (in kernal mode).
   Virtual memory manager tells disk manager about which page is missing.
3. Disk Managers communicates with device driver.
4. Device driver reads the page (through disk mechanism) and transfers required page to OS using DMA.
5. OS will try to load the page in one of the frames of user process.
6. If there is no empty frame to load new page then virtual memory manager has to initiate page replacement through which one of the pages in memory is replaced with required page. Here we have 2 cases:
   (i) page that is selected for replacement is not modified then we can directly overwrite it with new page. (1 disk operation)
   (ii) If it is modified (dirty page) then the modified page has to be saved in the disk. Later the new paged is loaded (i.e., 2 disk operations)
7. After the new page is loaded we need to update page table (Eg: changing V/I bit etc.)
8. Unblock the user process
   Now when the process is scheduled again, it generates the same VA that it had generated previously and continues its execution.

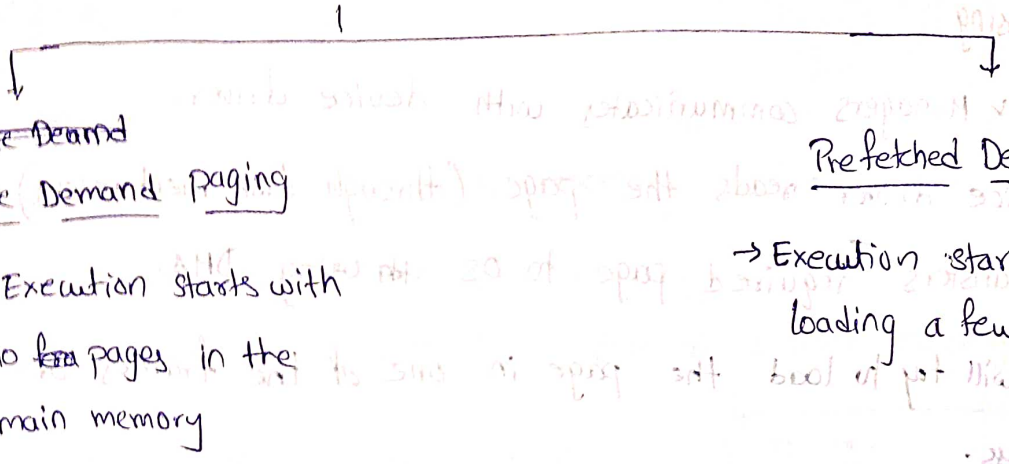
This is VM implementation through demand paging.

→ page fault service time is the time taken to service a page fault.

~~Hence we need to reduce~~
This service time reduces throughput.
So we need to reduce page fault rate.

If there is a TLB hit, then there is no page fault

→ Demand paging is of 2 types:



(By default we consider pure demand paging while solving questions)

(Q32) The size of virtual memory supported is eventually limited by size _____

Ans:
Size of disk (Secondary storage)

(Q33) ~~What~~

Note:
For real time implementation of virtual memory

PAS ≤ VAS ≤ Disk AS

Performance of Virtual Memory

(i) Temporal Issue: (time)

Assume

main memory access time = 'm'

page fault service time = 's' (s >>> m)

page fault rate = 'p' ($0 \le p \le 1$)

page hit rate = 1-p

EMAT of with Demand paging	= (1-p) m + p * s

(∵ s >>> m we don't take s+m)

In general pg. fault service time may range in milliseconds.

(14/16)

PFST = 10ms

Mem. AT = 1μs

page hit rate, $P = \frac{99.99}{100} = 0.9999$

$$EMAT = \frac{99.99}{100}(1) + \frac{0.01}{100}(10 \times 10^3)\ \mu s$$

$$= \frac{99.99}{100} + \cdots$$

$$= \frac{99.99}{100} + 1$$

$$= 1.9999\ \mu s \cong 2\mu s$$

(14/17)

page fault rate = $\frac{1}{k}$

page hit rate = $\frac{k-1}{k}$

Effective instruction time ~~EMAT~~ $= \frac{k-1}{k} i + \frac{1}{k}(i+j) = \frac{(k-1)i}{k} + \frac{i}{k} + \frac{j}{k}$

$$= i + \frac{j}{k}$$

H4/18

PFST = 8 ms (empty frame or unmodified frame)

= 20 ms (modified frame)

Mem. AT = 100 ns

p page to be replace is modified 70% of the time.

required EMAT ≤ 2000 ns

Let p be page fault rate

EMAT = (1-P)(M) + P(0.7(20) + 0.3(8))

= (1-P)(100) + P(14 + 2.4)

= 100 - 100P + 16.4P

⇒ 100 - 83.6P ≤ 2000 ns

EMAT = (1-P)(100) + P(0.7(20×10^6) + 0.3(8×10^6))

= 100 - 100P + P(14×10^6 + 2.4×10^6)

= 100 - 100P + 16.4P×10^6

⇒ (164×10^5 - 100)P ≤ 2000 - 100

⇒ (164000 - 1)(100P) ≤ 1900

$$P \leq \frac{19}{163999}$$

max acceptable page fault rate = $\frac{19}{163999}$

H4/19

Mem. AT = M (page hit)

= D (page fault)

for some process

· EMAT = X units.

page fault rate, P=?

$$X = (1-P)\,M + P(D)$$

$$X = M - MP + PD$$

$$\Rightarrow P(D-M) = X - M$$

$$P = \frac{X-M}{D-M}$$

Note:

Considering TLB & page fault rate the effective memory access time is

$$EMAT = x(c+m) + (1-x)\left[(1-P) * m + P * s\right]$$

$m + c$ ↳ (negligible)

x - hit ratio of TLB

P - page fault rate

(ii) page replacement policies:

Page reference string or reference string:

* set of successively unique pages refered in the given list of virtual addresses.

Eg: VAs: ⟨702; 755; 864; 084; 122; 560; 768; 934; 015; 125; 654⟩

Let page size = 100

Also all the above addresses are in ~~deca~~ decimal

page P_0 contains addresses 0 to 99

P_1 " " 100 to 199

⋮ ⋮

$$\text{page number} = \left\lfloor \frac{VA}{PS} \right\rfloor$$

$$\text{page offset} = VA \,\%\, PS$$

∴ Reference string for above VA's is

⟨7,8,0,1,5,7,9,0,1,6⟩

(successive page numbers are unique)

Every reference string has 2 parameters:

i) length (i.e. 10 in above example)

ii) no of unique pages (n) (i.e. 7 in above example)

Frame Allocation Policies

Assume we have

'n' no of process $(P_1 \dots P_n)$

demand of each process P_i be S_i no of frames

total demand for frames = $D = \sum_{i=1}^{n} S_i$

total frame available be: M $(D > M)$

frames allocated to process P_i be a_i

Eg: Let $n=5$, $M=50$

	S_i	a_i ①	a_i ②
P_1	5	10	3
P_2	20	10	10
P_3	44	10	22
P_4	18	10	9
P_5	12	10	6
D:	100		

① Equal allocation policy

Every process gets equal no of frame

$\therefore \frac{50}{5} = 10$ frames

② Proportional allocation

no of frames allocated is proportional to size

i.e., $\frac{S_i}{D} \times M$ no of frames for process P_i

③ x% rule

Every process is given x% of the frames it has demanded.

$\therefore$ This x is choosen based on main memory size and other factors.

Page Replacement Techniques (pure demand paging by default)

Consider for a process

reference string I: 7; 0; 1; 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

length of reference string, $l = 20$

no of unique pages, $n = 6$

1) FIFO:

Assume no of frames allocate for the process = 3

based on the time we loaded we replace

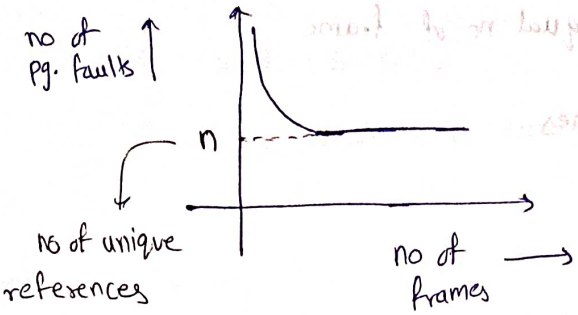
7	7	7	2	2	2	4	4	4	0	0	0	7	7	7					
	0	0	0	3	3	3	0	0	0	3	3	3	2	2	2	1	1	1	
		1	1	1	0	0	0	2	2	2	1	1	1	0	0	0	7	7	7

} frames

from above figure we have 15 page faults

$\therefore$ page fault rate $= \frac{15}{20} = 0.75$

If we increase no of frame allocated, page faults decreases



If no of frames allocated = 4

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

∴ 10 page faults

page fault rate = $\frac{10}{20} = 0.5$

Ref-String II : ⟨1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5⟩

$l = 12$

$n = 5$ (uniq ref)

FIFO

no of frame allocate = 3

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

∴ 9 page faults

Pg fault rate = $\frac{9}{12} = 0.75$

if no of frames allocated = 4

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

∴ page faults = 10

page fault rate = $\frac{10}{12} = 0.83$

In this example we got more page faults with more no of frames which is an anomaly.

This is known as Belady's anomaly

↳ with increase in no of frames to the process, the page fault rate increases

→ Beladys Anamoly is observed only in FIFO & FIFO based replacement.

11/05/20

2) Optimal replacement:

Optimal replacement says that replace that page which will not be used for longest duration of time in future references.

ref string: < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

3 frames:

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

∴ 9 page faults

4 frames:

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

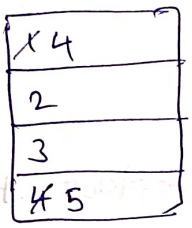
∴ 8 page faults

Ref string : <1,2,3,4,1,2,5,1,2,3,4,5>

3 frames :

∴ page faults = 7

4 frames :



∴ page faults = 6

→ Optimal replacement ~~gener~~ has the least page fault rate.

Drawback :

→ Optimal page replacement is not practically implementable since we can never know the future references.

→ However this can be used as benchmark to measure the performance of other algorithms.

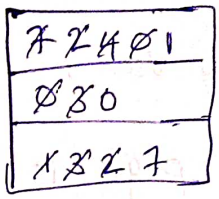
3) Least Recently Used (LRU) :

→ LRU replaces that page which hasn't been refered for the longest duration of time in the past.

∴ selection criteria : Time of reference

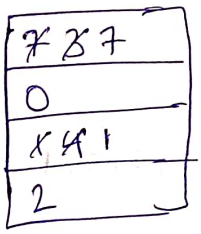
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 12

4 frames:



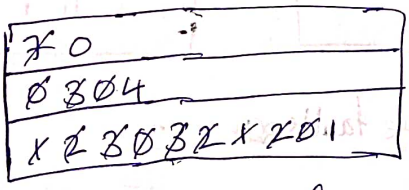
∴ no of page faults = 8

4) Most Recently Used: (MRU)

* Most recently used page is replaced

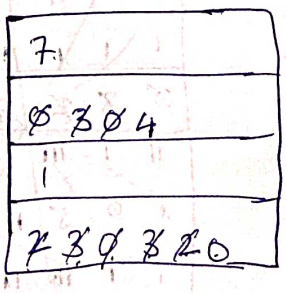
Ref string: <7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1>

3 frames:



∴ no of page faults = 16

4 frames:



∴ no of page faults = 12

5) Counting Algorithms

- LFU (Least Frequenty Used)
- MFU (Most Frequenty Used)

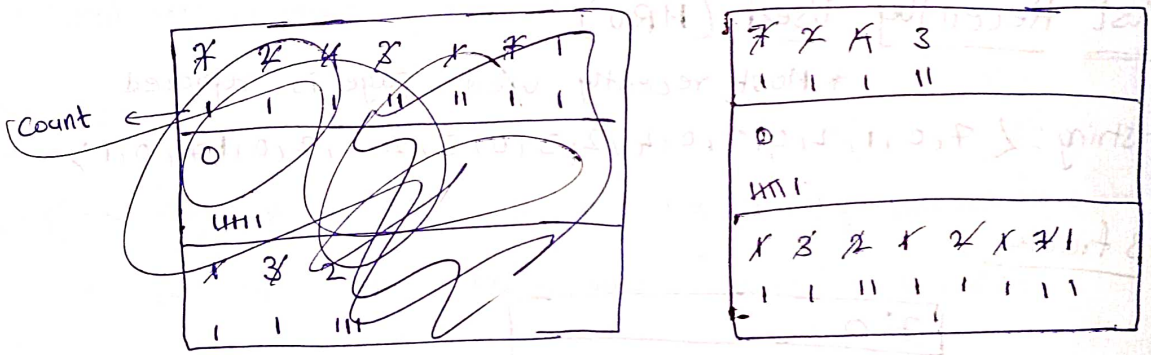
* Frequency means no of times the page is ~~refren~~ refered since it has been brought into memory. (i.e., if we load a page that has been loaded previously we start the count from 0)
* FIFO may be used ~~in~~ to resolve a conflict

3 ~~frame~~ :

Ref string : < 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 >

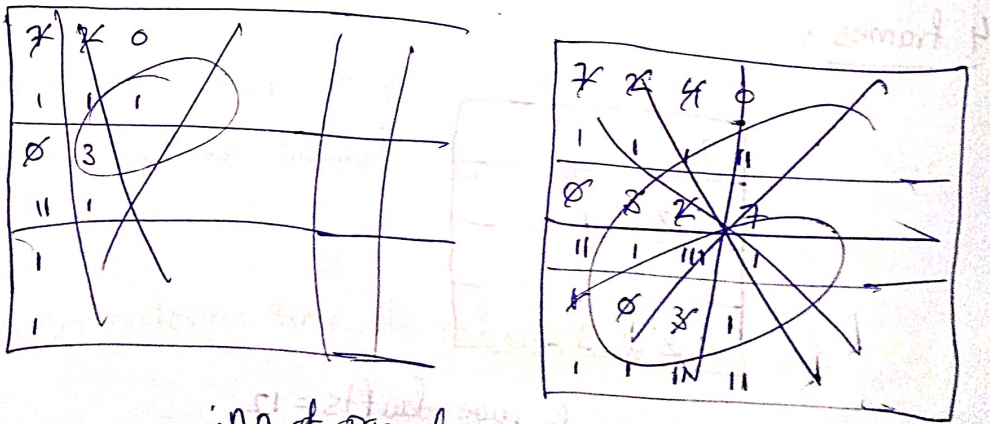
3 frames

LFU :



no of page faults = 13

MFU :

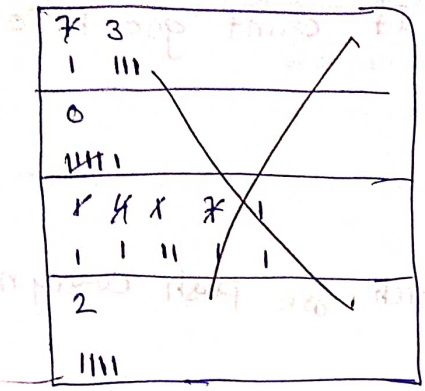


∴ no of page faults = 15

4 frames :

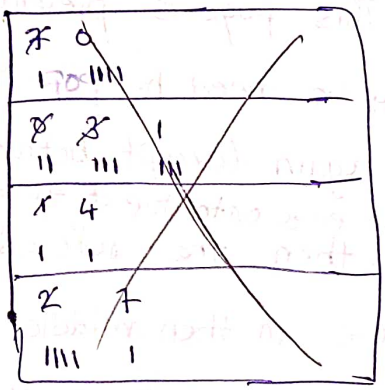
LFU :

Queue | 7 0 1 2 3 4 1



no of page faults = 9

MFU :



no of page faults = 9

12 is Ans

Note :

→ Practically LRU has been found to give less page faults (i.e., best performance & close to optimal)

Implementation of LRU :

* Here we need to monitor time of reference (TOR)

There are two methods to monitor time of reference

1) Counter method :

we maintain a counter register which is incremented for every memory reference. whenever we refer a page we assign the value of counter as time of reference for that page. Initial value of counter is 0.

Limitation:

* An n-bit counter can count only 2^n values after which count starts from 0. If count goes to '0' then the algorithm fails.

(ii) Stack method

→ Here we use a stack into which we push every new page we replace.

→ At the time of replacement, we need to replace the least recently used page. This page is present at the bottom of the stack. Thus we need to pop out all the elements and push them again (except bottom element) and finally push the latest page onto top of the stack.

→ If a page hit occurs, then the refered place might be present somewhere in then middle of the stack. This element has to be onto the top of the stack for which we need to perform push & pop operations again.

Advantage:

* Unlike counter method, it works good in any case.

Disadvantage:

* push & pop operation requires more time.

LRU Approximations:

* LRU approximations are the class of replacement techniques that pretend to work like LRU. i.e., they approximate to the behaviour of LRU.

we have 4 types of LRU approximations

i) Reference bit (R):

R { 0 : page not refered so far during present epoch
  { 1 : page is refered atleast once so far during present epoch.

Consider below page table

Pg No	frame No.	V/I	TOL	R
0	a	1	2	1
1	b	1	4	0
2	–	0	–	–
3	c	1	0	1
4	d	1	3	1
5	e	1	1	0

while implementing this algorithm we may not have TOL column

(TOL - Time of loading reference)

(Here in the Pg table Pg P_1 & P_5 are not refered in the present epoch)

* Time is divided into equal interval and each interval is called an epoch.

* when an epoch is finished all the 'R' values are cleared.

Algorithm:

→ * If page hit occurs, then set 'R' of the corresponding page table entry.

→ If page fault occurs,

Start searching ~~for R=0~~ from the first entry for R=0 and replace the first encountered entry's corresponding page.

→ If 'R' value is one for all the pages then the algorithm fails.

* To overcome this limitation we use next LRU approximation which is additional reference bits.

~~(ii) Additional bits~~

(ii) Additional Reference bits:

Here we have more than one reference bits, say 'n' indicating past 'n' references of past 'n' epochs

For example take n=8

	R_1 R_2 ..... R_7 R_8
P_i :	1 0 0 1 1 1 1 1
P_j :	0 1 1 0 1 1 1 1
R_k :	1 0 1 1 1 1 1 1

* Least significant bit corresponds to current epoch
* Most significant bit corresponds to most previous epoch

For above example if a page fault occurs P_j is replaced.

(∵ the 4 least significants of the 3 processes are '1'

However R_4 of P_j is 0 and R_4 of P_i & P_k are 1

∴ Replace P_j)

* After every epoch we need to left shift the reference bits by one bit.

* However this method also has a chance to fail when all the reference bits of all the processes are 1's. Anyway the probability for this is very less.

(iii) Second Chance (clock Algorithm) (Read from TB)

Selection criteria: (Time of loding & Reference bit)

→ The page with R value as 0 and least TOL (early loaded page among R=0 pages) is replaced.

The pages whose TOL is less than replaced page, their R value is set to '0'.

Also the arrival time of those pages is set to current time

→ when all R values are 1's, the earliest loaded page is replaced.

i.e., The algorithm, when all R values are 1, degenerates to FIFO

* Hence this algorithm (FIFO based) also suffers from Belady's anomaly.

(iv) Enhanced Second chance / Not recently used (NRU): (Read from TB)

Selection criteria: (Reference bit (R) & modified bit (M))

Modified bit (or) Dirty bit < 0: page is not modified since its loading
1: page is modified since the time of its loading

R	M	
0	0	(I)
0	1	(II)
1	0	(III)
1	1	(IV)

← This means the page is not refered in this epoch but is refered, and modified in some previous epoch

Page Buffering Algorithms:

(i) Maitaining free frame pool:

(i) choose victim; (ii) read page into one of free frame from pool; (iii) victim is written out later. (This will help restart process quicker). victim is added to free frame pool.

we may even maintain list of modified pages and write them on disk and reset their modified bit. This ↑ probability of find clean frame

we can even keep track of page present in frames of free frame pool. So that if same page is req it doesn't need to be read in. This all will help in ↓ penalty for replacing dirty page.

* If a page

then page with RM = 00 is replaced

if no such page RM = 01 is replaced

if no such page RM = 10 is replaced

if no such page RM = 11 is replaced

* The above search starts from the first entry.

H4/20

length of ref string : L

no of unique pages : K

allocated frames : Z

* In the worst case every reference can be a page fault

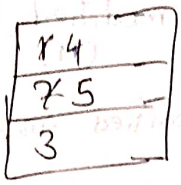
i.e., L

* Since it is pure demand page we must have atleast k page faults

~~In the best case we can have all page hits after loading the servicing the last page fault~~

Eg: Let $z=3$, $k=5$

Ref string: 1, 2, 3, 4, 5, 4, 5



$\therefore k$

In the best case the above could happen.

H4/21

Ref ~~string~~: $P_1, P_2, P_3, P_4 \cdots P_{n-1}, P_n, P_n, P_{n-1}, \cdots P_3, P_2, P_1$

↳ (This is not reference string since we have P_n twice successively)

After refering till P_n we will have

$P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)}$ in the frames

$\therefore$ The next k references $(P_n, P_{n-1}, P_{n-2} \cdots P_{n-(k-1)})$ will be hits

total references = $2n$

no of hits = k

$\therefore$ no of page faults = $2n-k$

H4/22 Ref string: ~~1, 2, 4, 5, 1, 2, 2~~, 2,

1, 2, 4, 5, 1, 2, 3.

$\therefore$ length of refstring = 7

no of frames allocated = 1

$\therefore$ 7 page faults

(H4/23) VAS = PAS = 2^{16} bytes = 64 kB

PS = 512 bytes = 2^9 bytes

PTES = 4 bytes = 32 bits

no of frames = $\frac{2^{16}}{2^9} = 2^7$

no of bits to store frame number = 7
V/I = 1
R = 1
Modified = 1
Page Protection = 3
} 13 bits

∴ 32 - 13 = 19 bits

19 bits can be used to store other information

no of pages = $\frac{2^{16}}{2^9} = 2^7$

size of page table = $2^7 \times 4 = 2^9 = 512$ bytes

Thrashing:

* Excessive / high paging activity
  ↓
  act of servicing page fault

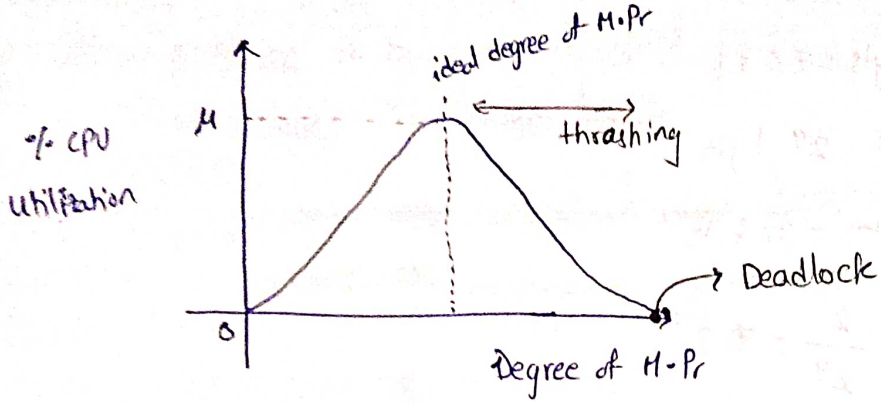
  i.e., high page fault rate

* Thrashing is undesirable
* Thrashing cause more processes to block

Cause of Thrashing

→ Lack of frames (i.e., Lack of memory)
→ High degree of multiprogramming
} Primary reasons for thrashing

(we also have some secondary reasons which we will explore later)



Thrashing Control strategies:

Prevention

* Prevention is done by controlling degree of multiprogramming. (Controlled by long term scheduler)

Detect & Recover

Detection : * High degree M.Pr & low CPU utilization
=> thrashing
* High paging (high disk utilization)

The above 3 conditions confirms the presence of thrashing

Recovery: Decrease the degree of M.Pr
(This done by suspending the processes)

Secondary reasons for thrashing:

→ Page replacement policy

→ page size:
page size must be large to reduce page faults.
It is because with large PS we have less pages.
(Large PS => less pages => less page faults)

→ Programming techniques & Data structures used by programmers. This fact is explained through below case studies.

Case study I:

```
integer A[1...128][1...128];
```

1) for i←1 to 128
    for j←1 to 128
        A[j][i]=1;

2) for i←1 to 128
    for j←1 to 128
        A[i][j]=1;

Assume we store the array in row major order

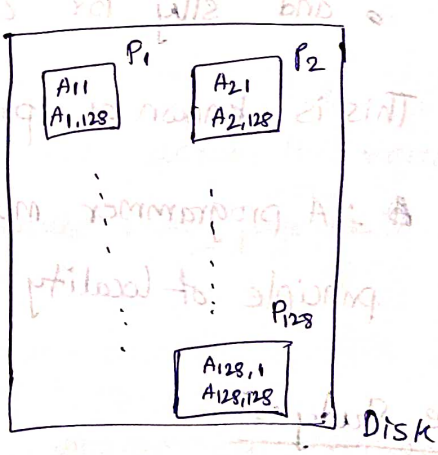
also let page size = 128 words

and each element of array is one word

⇒ each row is stored in one page exactly

Assume we are using pure demand paging and replacement technique FIFO.

→ Assume the OS has allocated 127 frames to the process.



page faults with 1st program:

→ A[0][0]. A[1][1] .... A[128][1]

Here every reference causes a page fault

i.e., 128 page faults.

Sily for 2nd column we get 128 page faults

In this way

total no of page faults = 128 × 128

Page faults for 2nd program :

→ A[1][1], A[1][2] ... A[1][128]

page fault — page hit

∴ 1st row causes one page fault

Silly we get 1 page fault for 2nd row

∴ total no of page faults = 128

→ ∴ 2nd program is 128 times faster than that of 1st program

Note :

* If we use row major order to store 2d array then it is better to access in that order
  - and silly for column major order.

This is known as principle of locality

* ∴ A programmer must use data structures that ensure principle of locality.

Case Study - II

* If we consider array & linked list,

  array allocation is contiguous

  linked list allocation is non-contiguous.

  ∴ linked list may cause more page faults.

* The elements of array obeys principle of locality.

Case Study III

Linear Search vs Binary Search

→ In demand paging, b binary search causes more page faults because we don't access elements contiguously.

whereas in linear search we search in linear contiguous way causing less page faults and obeying principle of locality.

Case Study - IV

Stack:

Stack operation which are push & pop are performed at the same end.

∴ less page faults

Hashing:

Hashing distributes key across the memory and is more likely to cause more page faults.

Note:

→ In general a Data Structure or a programming technique in demand paging is said to be good iff it obeys the concept of locality of reference.

Working Set Strategy

* The objective of working set strategy is to control thrashing and utilize memory effectively.
* Working set strategy is based on principle of locality of reference.

Consider below program

```
main()//10kB        f()//5kB        g()//30kB       h()//3kB
{                   {               {               {
   :                   :               :               :
   f();                g();            h();            printf(); //25kB
   :                   :               :               :
}                   }               }               }
```

Total program size = 73 kB

Assume page size = 1 kB

Assume the process has been allocated 35 frames

* This is static allocation.

→ As the process executes it changes from one locality to other locality (i.e., from one function to other function). When process is in main 10 frames would be enough. When it is executing in f(), 5 frames would be enough.

→ Thus allocating frames based on the locality in which the process is executing would reduce thrashing and utilizes memory more effectively.

→ To meet this requirement we use dynamic frame allocation technique.

* The working set strategy's working is based on the locality of reference.

It estimates the locality in which the process is running and no of frames are allocated accordingly.

* Estimating the size of locality:

Consider the reference string for process P_i

$\Delta = 10$

P_i : ⟨ 7, 0, 1, 2, 3, 0, 2, 16, 18, 22, 18, 16, 35, 38, 39, 34, 35, 37, 38, 36, 34, 36, 37 ⟩

Assume these references are generated in time t. After time t, we need to know the locality in which the process is executing and size of locality.

* observing the patterns we can know localities & ⊕ size of localities.

* to estimate the size of locality we define a parameter called working set window (wsw) at time t for a process

Working set window$_i^t$ (wsw) = { set of unique pages refered in the past Δ references }

where Δ is an integer.

For example take Δ = 10

⇒ wsw_i^t = { 34, 35, 36, 37, 38, 39 }

⇒ $wsw\text{-}size_i^t = 6$

↓

size of locality is 6 pages

∴ 6 is the demand

Framework:

→ Let 'n' be no of processes

→ s_i : Demand for frames by process P_i (Estimated as shown in the previously example)
(i.e., wsw size$_i$)

→ total demand, $D = \sum_{i=1}^{n} s_i = \sum_{i=1}^{n} wsws_i$

→ Available frames = M

Case (i):

$D \cong M$

i.e., Balanced & no thrashing

Case (ii)

$D < M$

i.e., No thrashing & we can further increase deg of M.pr

Case (iii)

$D > M$

i.e., Thrashing

∴ Working sets strategy helps control thrashing and utilize memory better.

Note:

* The success of Working Set Strategy depends on the value of Δ.

* If Δ is too small ⇒ more page faults.

* If Δ is too large

⇒ we also hold pages of previous locality and this is ineffective use of memory

4/24

page ref : c c d b c e c e a d

$\Delta = 4$

$WST_0 = \{a, d, e\}$

0

i.e., $\{e, d, a\}$ (-2, -1, 0)

∴ ref string: e d a c c d b c e c e a d

-2 -1 0 1 2 3 4 5 6 7 8 9 10

⇒ 5 page faults

t_0 : <e d a> : 3

t_1 : <e d a c> : 4 → At time t_1 'c' is refered since wsw at t_0 doesn't have 'c' it causes page fault

t_2 : <d a c> : 3

t_3 : <a c d> : 3

t_4 : <c d b> : 3 → Silly at time t_4 'b' is refered since 'b' is not present in wsw at t_3 a page fault is generated

t_5 : <d b c> : 3

t_6 : <d b c e> : 4

t_7 : <b e c> : 3

t_8 : <c e> : 2

t_9 : <c e a> : 3

t_{10} : <c e a d> : 4

Avg no of frames needed = $\frac{32}{10} = 3.2$ frames

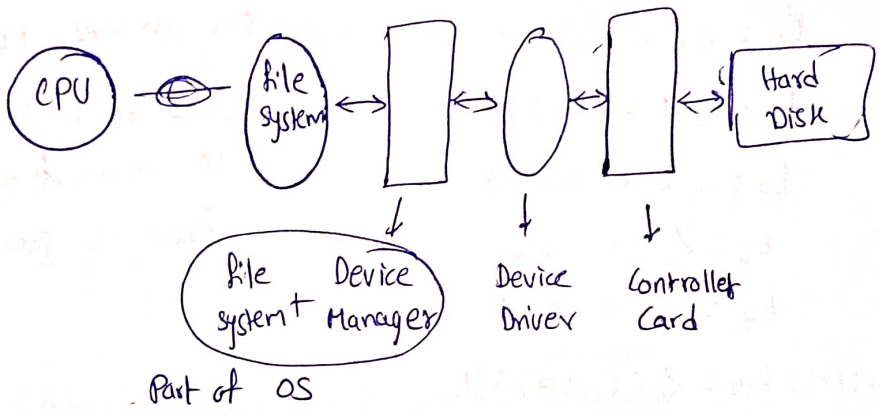
File System & Device (IIO) Management

→ File system is slow for managing secondary storage devices (Disk)

* CPU is an electronical device & Hard disk is electro-mechanical device and hence they are ~~are~~ architecturally incompatible to communicate directly. Hence for the purpose of the communication we use an interface known as Controller card or interface.

* To drive this controller we need device driver. This driver is device specific.

* To manage the device driver we need device manager



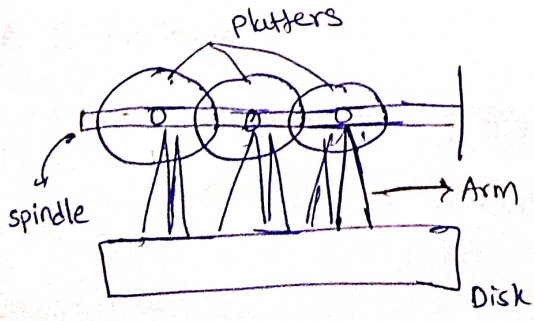
physical structure of Hard-disk:

* Hard-Disk ~~a~~ - collection of platters
* Every platter has 2 surfaces
* Every surface is ~~dev~~ divided in tracks
* Every track is ~~dev~~ divided into sectors

tracks are ~~mapped~~ counted from outermost to innermost.

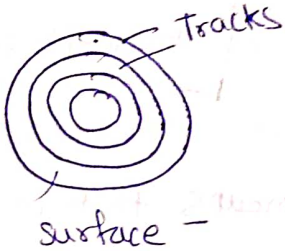
→ logical counting is done as

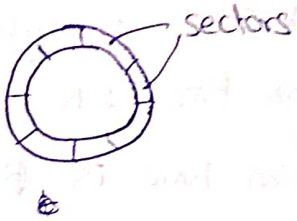
0th sector of outermost surface of outer cylinder then other tracks of same cylinder. Then inner cylinder



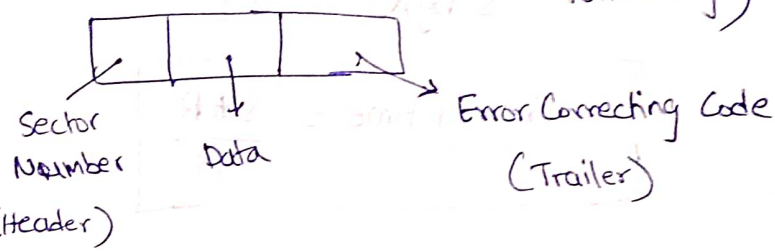
Every ~~platter~~ surface has a read-write head ~~and~~
i.e., a platter has two read-write heads

→ The disk assembly can move forward & backward and the spindle can rotate either in clockwise or anticlockwise direction





Each sector has 3 parts: (Created at the time of low level formatting)



→ Same track of all the surfaces forms a cylinder
i.e., no of cylinders = no of tracks on the surface

I/O time to read a sector/block:

seek-time:

It amount of time taken to move the arm from to the desired track.

Rotational latency (or) Latency time:

The amount of time taken to bring the required sector under read write head.

If rotation latency is taken as $R/2$
R — time for one rotation.

transfer time:

It is amount of time taken to transfer one sector from disk to controller buffer.

Track Track time:
time required for the R/w head to move from one track to the adjacent track

Calculating transfer time:

Assume

sector size : s bytes
track size : x bytes
rotation time : R ms

Since rotation time is R, it means that in R ms we can read entire track i.e., x bytes

x bytes — R ms
s bytes — ?

$\therefore$ Transfer time = $\frac{s * R}{x}$

* Transfer time is comparitively very less than seek time and rotational latency.

Disk Bandwith:
Amount of data transfered divided by time b/w first request for service and completion of last transfer

(H4/3) Avg seek time = 30 ms
Rotation time = 20 ms
track size = 4kB
Prog. size = 64 kB
PS = 4kB

The program is stored on the disk

no of pages = $\frac{64}{4}$ = 16 pages

time to load

Random distribution :

time to load a program = (time to load a page) * 16

~~transfer~~ 32 kB — 20 ms

4 kB — ?

$\frac{4 \times 20}{32} = 2.5$ ms

∴ time to transfer one page = 2.5 ms

time to load a program = $\left(30 + \frac{20}{2} + 2.5\right) \times 16$

= 680 ms

50% Contiguous :

i.e., 32 kB is contiguous

i.e., 50% is on the same track

time to load non-contiguous = $\left(30 + \frac{20}{2} + 2.5\right) \times 8$

= 340 ms

time to load contiguous pages = $\left(30 + \frac{20}{2} + 20\right)$

- $\frac{20}{2}$ → one rotation
- 20 → transfer time of 8 pages (8 × 2.5)

= 60

∴ total time = 400 ms

∴ % of time saved = $\frac{680 - 400}{680} \times 100$

$= \frac{280}{680} \times 100$

$= \frac{7}{17} \times 100 \cdots = 41.1\%$

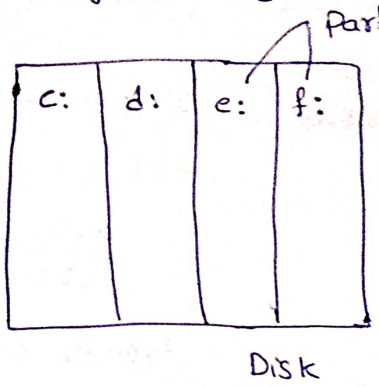
20/08/20

Logical structure of Disk: (Formatting Process)

~~* Disk is divided into logi~~

* Disk is logically divided into partitions. ~~or volumes~~

~~* A partitions containing OS is called volume.~~



* fdisk is a utility to create partition

  df (disk framangling) is also used to create partitions.

~~Partitions~~

Partitions

A disk containing os is called System disk

↓

(or) boot disk

Primary partitions

* These partitions are bootable

  i.e., we can store OS & data

Extended partitions

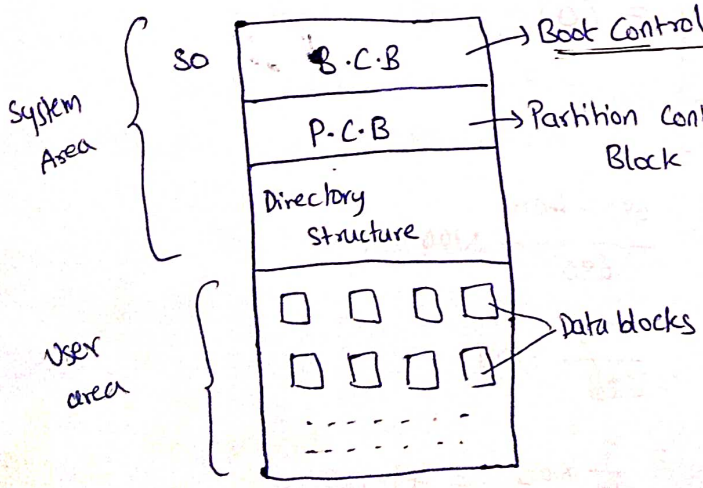
* These are logical drives
* They store only data (no os)

* A partition which is not formatted is called raw partition.

  I/o to this partition is called raw I/o.

Any entity of disk that contains a file system is called volume

Partition Structure (created cuz of formatting)



Boot Control Block

↳ A block which is meant for storing booting program of O.S. If we don't have booting program, then the block should be left empty.

Partition Control Block

→ Partition Control Block contains information about partion (like size of partition, no of block of partition in use etc.)

In unix file system, PCB is known as Super Block

In NT file system, PCB is called Master File Table

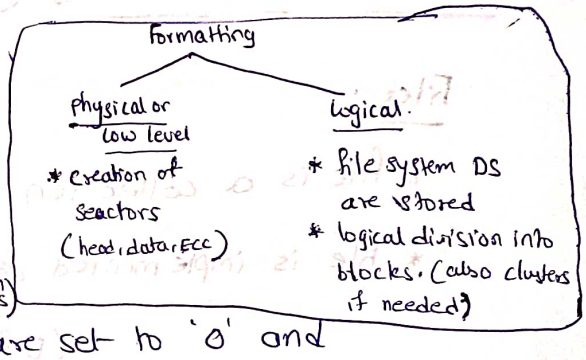
→ formatting divides disk into blocks.

→ Anything that gets stored on disk will be stored in terms of blocks. All I/o is performed in terms of blocks.

Block size ↑ ⇒ Internal Frag. ↑ & Data tranfer rate ↑

Booting steps:

In a primary partition which contains os, we have a Master Boot Record (MBR) in a fixed address location.



* MBR has two things:
  - (i) Boot loader (code)
  - (ii) Partition table (It contains table listing partions & flags indication the partion with os)
* MBR is placed on first sector of the disk.
* when we switch on PC, all register are set to '0' and Program counter (PC) is set to fixed address (address of Boot ROM)
* i.e., address of program BIOS (Basic Input Output System)

BIOS:

Bootstrap

Step1: Program POST (Power On Self Test) is run initially
this program checks the functionality of all the component and sees if there is any error.
If there is any error, error is reported and POST is terminated.

Step2: RAM test is done to see if memory is alright or not.

Step3: Device initializations

Step4: Identify booting source

Step5 : MBR is loaded into memory

If there are multiple OS's present MBR allows user to the select the required OS.

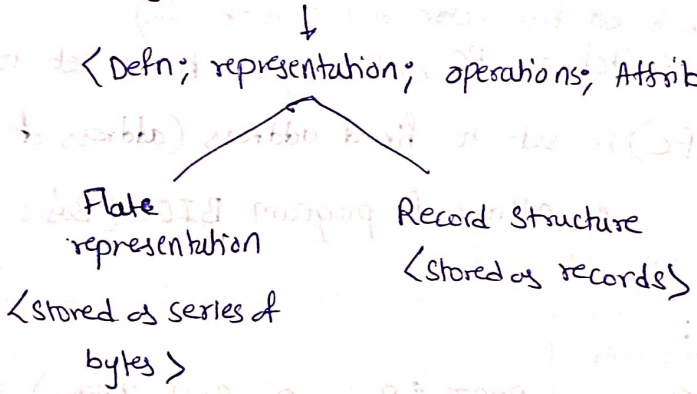
After the selection, boot loader transfers the control to that particular partition.

* Now the Boot Control Block comes into main memory.
* Now this BCB initializes various modules of the OS (i.e., memory managers, interrupt handler etc.)
* The last program that is executed at the end of the booting process is shell or GUI.

Files vs Directories <File system interface to users>

Files :

* file is a collection of logically related records of an entity.
* file is implemented as an ADT



→ Hierarchical structure is an extension of record structure (Eg: Trees, B-trees etc.)

operations:

→ Create
→ open
→ read/write/update/truncate
→ move/copy/rename
→ close
→ delete

These operations are supported through
(i) Command interface or GUI
(ii) System calls

Attributes:

general attributes:
- → File id, file name
- → type (Eg: user file, os file, executable file etc.)
- → size
- → owner
- → Date & time stamps (like date of last access, date of creation etc.)
- → link count (used for sharing. link count is used to say no of processes sharing the file)
  - ↳ link can be either hard link or soft link

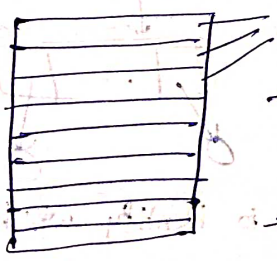
location attributes:
- → path (location on disk)
- → Blocks in use

→ These attributes are maintained as table which is known as F.C.B (File Control Block).
- ↳ known as inode in unix
- ↳ known as directory entry in windows

Directory:

* Directory is a collection of files
* Directory is also a file (special file)
  - ↳ contains data about file (metadata)
* Every file system has a root directory.

Abstract view of Directory



Directory

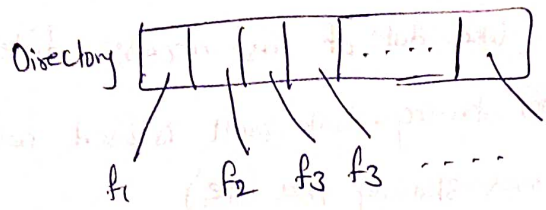
- One entry per file.
- → Each entry contains attributes of the corresponding file
- → Each entry is called directory entry

* One operation that can be performed only ~~dirrc~~ on directory, but not on a file, is traversing.

Directory structures

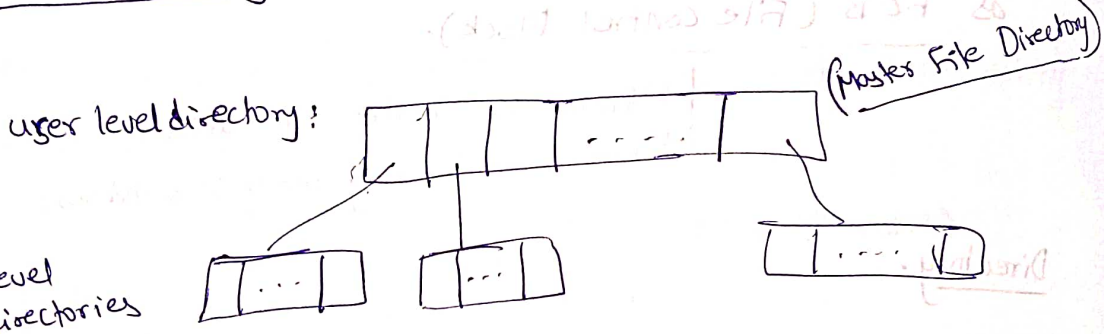
(i) Single level directory:

* only one directory is maintained for entire file system



* Advantage: simple to implement
* Drawback:
  - More searching time
  - Name of files must be different

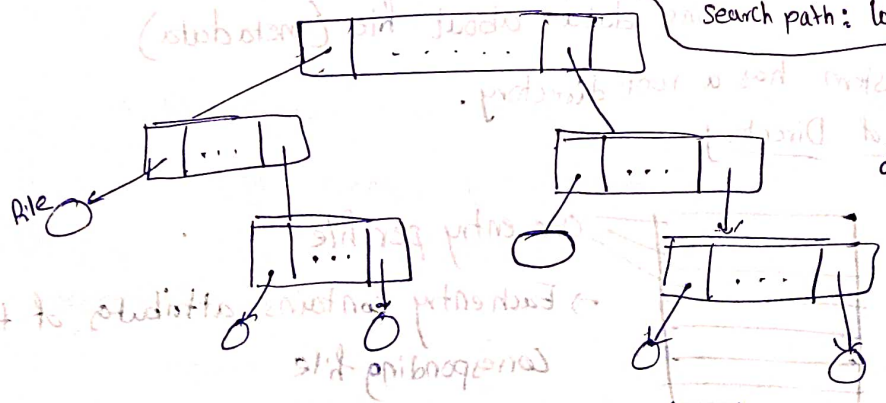
(ii) 2-level directory structure



Disadv: one UFD is completely isolated from other UFD. So co-operation btw users is not possible. To give access (allow sharing) one user has to be given ability to name a file with other users UFD.

Search path: local UFD - special user directory (contains system files)

(iii) Tree Structured Directory:



~~disadv~~: File sharing can be done by specifying path names. ~~Sharing directly~~ is ~~not possible~~.

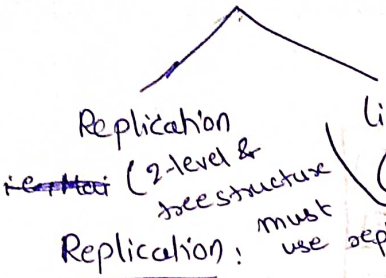
→ A bit is used in directory entry to identify whether it is file or subdirectory.

→ Here path name can be of 2 types: Absolute path (begins from root); relative path (begins from current directory)

→ Deletion of directory (2 ways):
- allow to delete only empty directory. So we need to delete all the files & subdirectories 1st.
- Provide an option to user like in the case of UNIX.

(iv) DAG (Directed Acyclic Graph Directory Structure)

* This is used for the purpose of file sharing
* Sharing can be done in two ways:



→ In directory entry we also save links & mark it as links. At the time of traversing, the links are ignored.

→ With files being shared by creating links, a file may have 2 diff absolute paths. So distinct file name may refer to same file.

Replication:

* maintaining seperate copy for each user
* requires more memory.
* may lead to inconsistency

Links: (Hard link or Soft link or Symbolic link)

* Every user has a pointer to same file.
* we maintain link count to keep track of no of users sharing the file.

→ with links added to the tree structure it seems as DAG

→ DAG has no loops and hence searching never goes to infinite loop.

→ However general graph directory structures allow loop and hence searching mechanism has to ensure that it does not go into infinite loop

Note:

→ Since Directory is a file, we can also perform same operations on directory.

→ create, open, read, write, delete etc.

→ Traverse

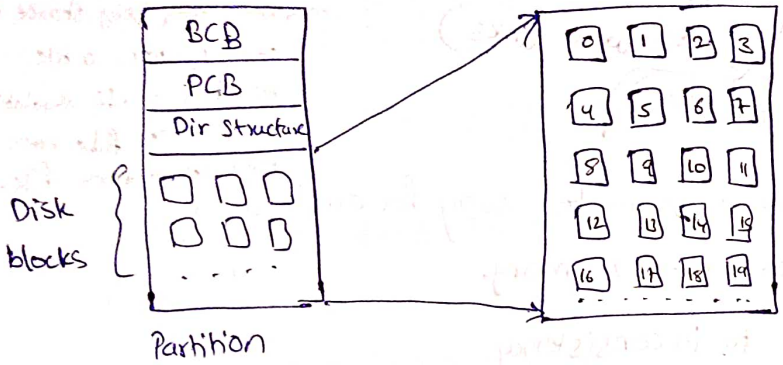
this operation can be performed only on directory but not a file.

→ Attributes of directory are also mostly similar to the attributes of files.

File System Implementation

1) Allocation Methods:

i.e., Disk space allocations



Every Disk block has two attributes

(i) DBA (Disk Block address) (in bits)

(ii) DBS (Disk Block Size) (in bytes)

For example,

if we have 32 disk block then

size of DBA is 5 bits

Eg: DBA : 16 bits DBS : 1kB (This will be the working example for below allocation methods)

Maximum file size possible on this disk

= no of disk blocks x size of each disk block

$= 2^{16} \times 2^{10} = 2^{26} = 64$ MB

> Max file size possible $= 2^{DBA} \times DBS$

(i) Contiguous Allocation Method:

Assume we have a file of size 13kB

∴ No of blocks required, $N_{Blocks} = \frac{13kB}{1kB} = 13$ blocks

Now we need to seach for 13 contiguous blocks in the ~~memo~~ disk.

~~In~~ the below figure ~~is~~ is directory structure

other attributes

File Name	. . . . .	First DBA	size in no of blocks
K.C		6	4

Any of first fit, best fit & worst fit may be used for allocation

↓

i.e., we have ~~6 to~~ 4 contiguous blocks starting from 6th block.

Looking ~~attre~~ at directory structure we can search for contiguous free blocks.

performance issues:

* Suffers from internal fragmentation (in last block)

* Suffers from external fragmentation
  (∵ free blocks may be scattered accross the disk)

* If we need to increase file size, ~~it is inflex~~ it is not easy.

* Type of access

  * → Supports both sequential & random access

  ↓

  advantage: faster access

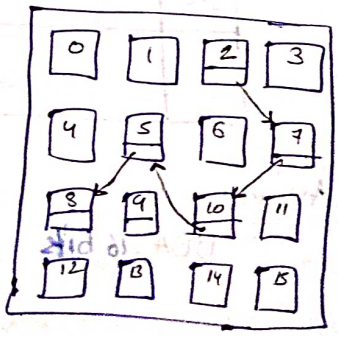
* To solve ~~th~~ problem of EF, we may use compaction.

ii) Non-Contiguous allocation/Linked allocation:

↳ This must be done with the support of another disk.

Directory Structure

File name	. . . . .	First DBA	Last DBA
K.C		2	8
⋮	⋮	⋮	⋮



→ Linked list is created among blocks ~~is~~ used by the file

Every block of a file contains pointer to its next block

Performance Issues

* Suffers from internal fragmentation
* No external fragmentation
* Increasing size is flexible
* Disadvantages:
  * Type of access
    → only sequential (Hence slow)
  * pointer consume disk space (overhead)
  * Vulnerability of pointers
    i.e., If a link gets breaked the file gets truncated
    That is why we use last DBA in directory structure to see if the file is completly accessible or not.

Note:

→ Also to storing last DBA is useful when we need to increase the file size.

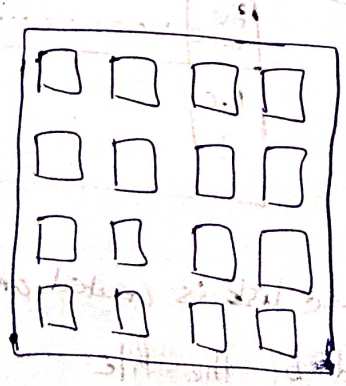
* (iii) Indexed Allocation

* Combination of Contiguous & Non-Contiguous

Directory Structure

file name	----	Index block
K.C		23
J.C		15
:	:	:

→ Every file is associated with an index block
→ Index block is also one of the disk blocks.



Assume
DBA : 16 bits DBS : 1KB

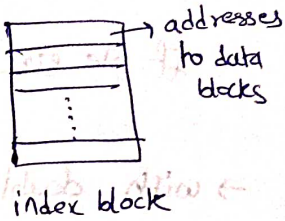
* Index block contains addresses of data blocks that are in use by file

Max file size possible with one index block

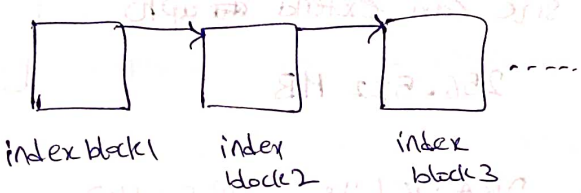
= no of entries in an index block * size of block

= $\frac{1KB}{2B}$ * 1KB => 512 kB

∵ 16 bits to address a disk block

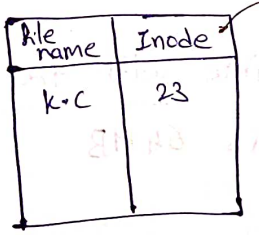


* Now if the we file size is more than 512 kB we can have more than one index block as linked lists



Case Studies

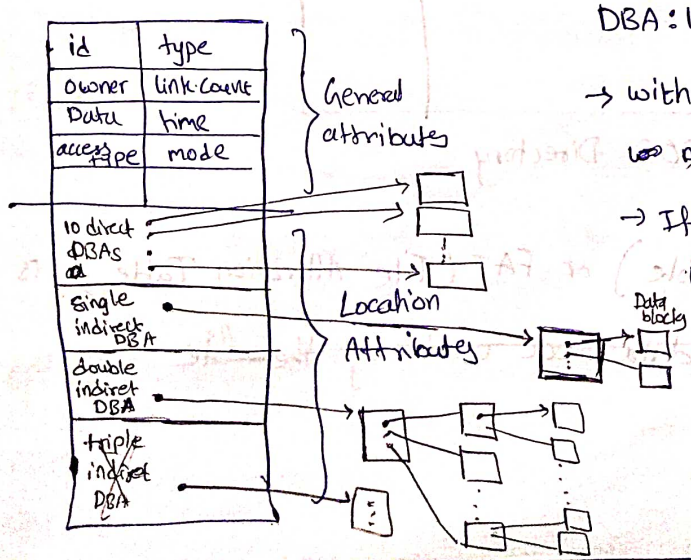
I. Unix | Linux :



→ Index (File Control Block)

$ls -i is used to obtain inode numbers of files.

*** I-node structure :



DBA : 16 bits DBS : 1 kB

→ with 10 direct DBAs file size will be 10kB

→ If file size is more than 10 kB we use single indirect DBA which point to a block that contains other data blocks addresses.

→ Now the file size that can be accessed is 512 kB with indirect DBA.

∴ now file size can extend upto 522kB

If file size is more than 522kB we go for double indirect DBA

→ with double indirect DBA we can access a file size upto

$512kB \times 512 = 2^{19} \times 2^9 = 2^{28}$

i.e., 256MB

This is known as Multi-level index type of allocation

Now the total file size can extend upto

256.522 MB

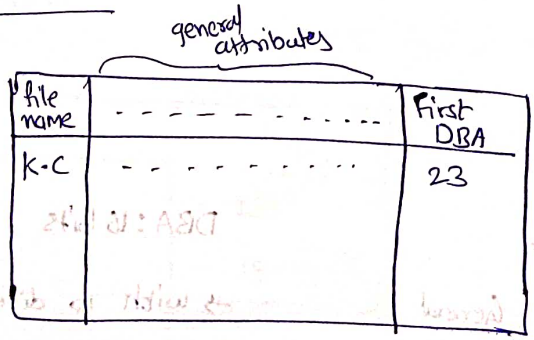
However having DBA: 16 bits DBS: 1kB

the maximum size itself possible is $2^{16} \times 2^{10} = 2^{26} = 64MB$

∴ Finally the maximum file size possible with the given inode structure is 256.522 MB

and the maximum file size possible with the given inode structure on the given disk is 64 MB

II. DOS/WINDOW



DOS Directory

* MFT (Master file table) or FAT (File Allocation Table) is maintained

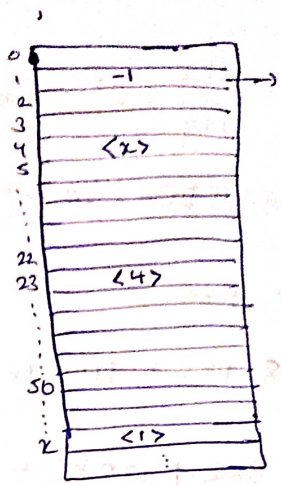
~~to obtain other data block used by the file~~

→ No of entries in FAT = no of data blocks

→ FAT entry contain address (DBA) of next data block in use by file

In the last block we may use special number like '-1' to ~~end~~ indicate end of file.

23 → 4 → x → 1
(end)



* This is known as tabular linked allocation

< we are indirectly creating a linked list in array (FAT) >

Hs/8 In unix it is not direct index we are using.

we are using multiple levels of index which is an extension to indexing

∴ opt (d)

Disk Bandwidth:
Amount of data transfered divided by the time blw the request for service and completion of the transfer

Note:

→ when DBS increases, Int. Frag. increases ⇒ poor disk space utilization

→ when DBS increases, disk throughput increases.

Disk throughput:

Amount of data that can be accessed per unit time.

~~If disk~~ If DBS is more, with one seek time & one latency time and one transfer time we will be able to read more data.

Hs/9 opt (a)

H5/5

$$DBA = \frac{1KB}{128} = 2^3 = 8 \text{ bytes}$$

$$\Rightarrow DBA = 64 \text{ bits}$$

(i) Max file size

$$= 8 \times 1KB + 128 \times 1KB + 128 \times 128 \times 1KB + 128 \times 128 \times 128 \times 1KB$$

$$= 8KB + 128KB + 16MB + 2GB$$

$$= 2GB + 16.136\ MB$$

$$= 2.016\ GB$$

(ii) 8 bytes = 64 bits

(iii) Disk size = $2^{64} \times 1KB$

$= 2^{74}$ bytes

∴ possible

H5/7

DBA = 8 bytes = 64 bits

DBS = 2^7

~~max file size = 8+~~

no of address per disk block = $\frac{128}{8} = 16$

max file size = $(8 + 16 + 16*16)$ blocks

$= (2^3 + 2^4 + 2^8) \times 2^7$

$= 2^{10} + 2^{11} + 2^{15}$

$= 1KB + 2KB + 32KB$

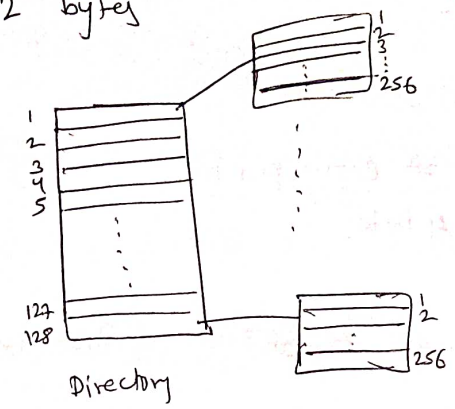
$= 35\ KB$

H5/6

H5/10

H5/

H5/6 DBS = 2^{12} bytes



$\therefore$ max file size = $(128 \times 256) \times 2^{12}$

$= 2^{27} = 128$ MB

H5/10

FAT entry = 4 bytes

$\Rightarrow$ DBA = 4 bytes = 32 bits

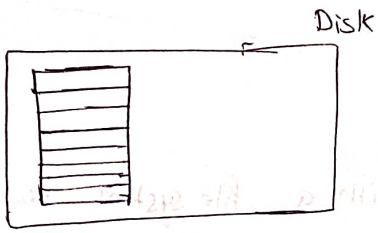
~~no of disk blocks = 2^{32}~~

~~DBS = 10^3 bytes~~

no of disk blocks = $\frac{100 \times 10^6}{10^3} = 100 \times 10^3$

H5/10

$N_{blocks} = \frac{100 \times 10^6}{10^3} = 100 \times 10^3$



total size of FAT = $100 \times 10^3 \times 4$

$= 0.4 \times 10^6$

$\therefore$ remaining disk size = $(100 - 0.4) \times 10^6$

$= 99.6$

$\therefore$ Max file size = 99.6

H5/11

DBS = 4 KB = 2^{12} bytes

DBA : 10 bits

Disk block 2, 3 : 32-bit entry per file

i.e., 4 bytes

(i) two disk blocks are used as directory

i.e., 8 kB directory

2^{13} bytes

$$\text{no of files} = \frac{2^{13}}{\text{size of entry per file}} = \frac{2^{13}}{4} = 2^{11} = 2\text{k files}$$

(ii) 10-bit entry

$\Rightarrow$ 1024 blocks.

Block 0, Block 1, Block 2, block 3 are used for formatting

So we have 1020 more blocks left

$\therefore$ Max file size = (1020) * 4 kB

= 4080 kB

= 4.08 MB

Free Space Management:

Consider a disk of size 40 MB

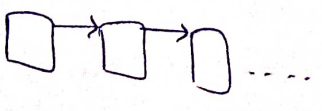
Assume the disk is formatted with a file system having

DBS: 1 kB & DBA: 16 bits

$$\text{No of blocks} = \frac{40 \times 2^{20}}{2^{10}} = 40\text{k blocks}$$

1) Free linked list:

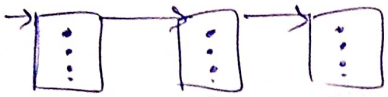
All free blocks are linked



If we want to create a file with 'k' blocks, then 'k' blocks are deleted from the linked list

2) Free list / Grouping :

a linked list of data block is maintained. Each block contains addresses of free blocks.



If all 20k blocks are free then

no of entries in blocks in list

$= 20k \times 2$

$= 20 \times 2^{10} \times 2 = 40$ kB

no of nodes required to store these entries

$= \frac{40\text{ kB}}{1\text{ kB}} = 40$ blocks

3) Bit Map Method :

we associate a binary bit with each block

bit : 0 : block is free

1 : block in use

In the previous example, the size of bit map is 20k bits

$\therefore$ no of blocks required for bitmap $= \frac{20k}{8} = 2.5$ kB

$\therefore$ 3 blocks

Read & Counting from Galvin Pg: 563

Note :

* bitmap requires less space than free list. (In most of the cases)
* Free list contain addresses of free blocks and heance search is easy. In bit map searching takes more time.

H5/12

A) (i) Disk size = BX

(ii) Max possible disk size = 2^D blocks

$= 2^D * X$

(iii) $B \leq 2^D$

B) size of bitmap = B bits

= B blocks = B bits

size of free list = F * D F * D bits

$\therefore F * D < B$

H5/13

a) 1111 1111 1111 0000 : FFF0

b) File A is deleted

1000 0001 1111 0000: 81F0

c) 1111 1111 1111 1100 : FFFC

d) 1111 1110 0000 1100 : FE0C

Disk Scheduling / IO Scheduling / Device Scheduling:

Need for IO scheduling:

* Every device has its own device queue.
* Now we need to schedule the processes in the device queue for the device.

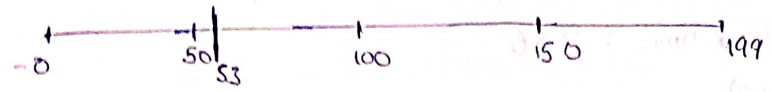
Here device may be disk, or any other Ilo device.

Disk Scheduling strategies:

Process Req's: 98, 183, 37, 122, 14, 124, 65, 67 ... track numbers.
P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8

Assume all the above reqs are available at time t=0

let no of tracks in disk = 200



let initial position of ~~head~~ ~~track~~
r/w head = track 53

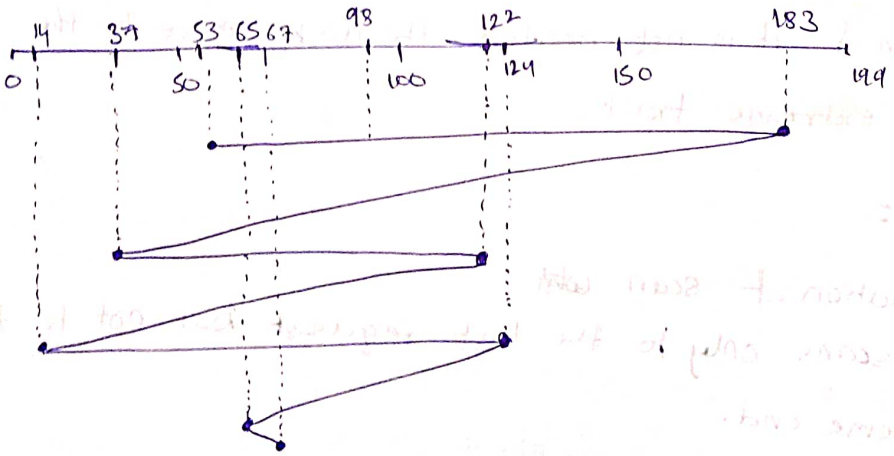
* Minimizing seek time is the objective.

1) FCFS:

~~98 to~~

~~53 to 183~~ , 1

53 to 98, 98 to 183, 183 to 37 ---- 65 to 67



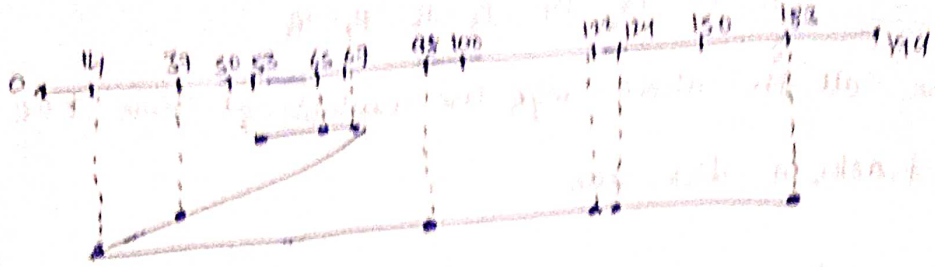
total no of seeks = (98 - 53) + (183 - 98) + (183 - 37) + (122 - 37)
+ (122 - 14) + (124 - 14) + (124 - 65) + (67 - 65)

= 640

Avg no of seeks = $\frac{640}{\text{no of processes}} = \frac{640}{8} = 80$

2) Shortest Seek Time First (SSTF):

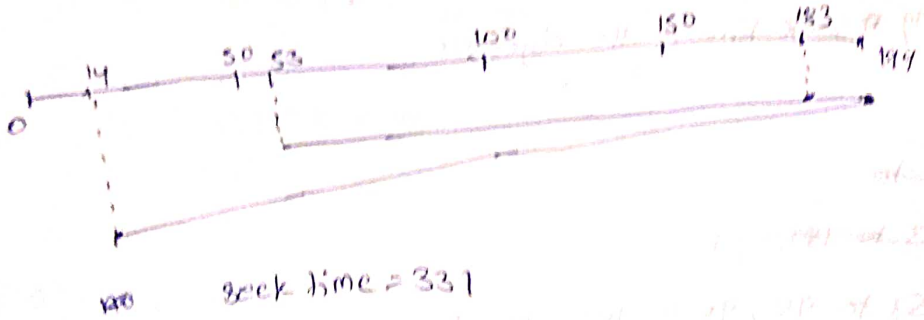
closest (closest track next) (closest cylinder next)



Seek time = 236

* Starvation is present.

3) Scan / Elevator Algorithm



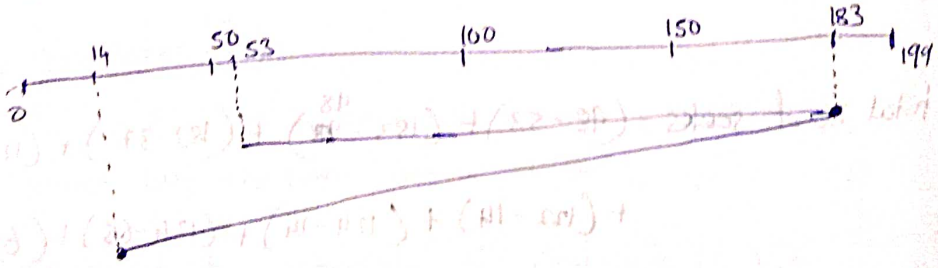
Seek time = 331

Drawback:

Even if it is not needed the disk move to the extreme tracks

4) Look:

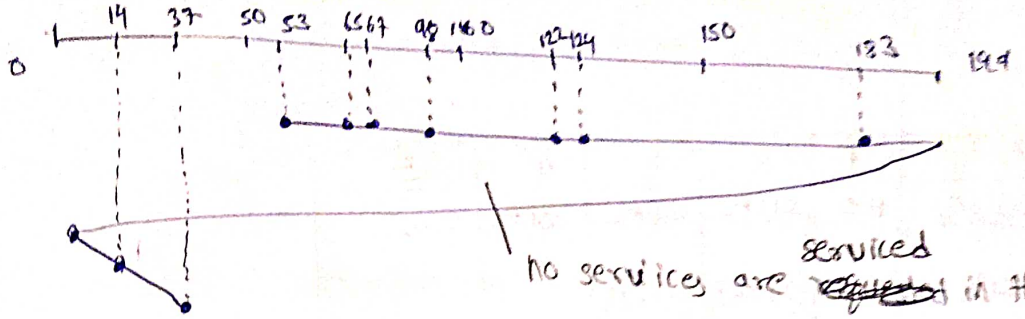
* Variation of scan
* It scans only to the last request but not to the extreme end.



5) C-Scan:

(Circular scan)

After going to last track (199) we immediately start from '0'.



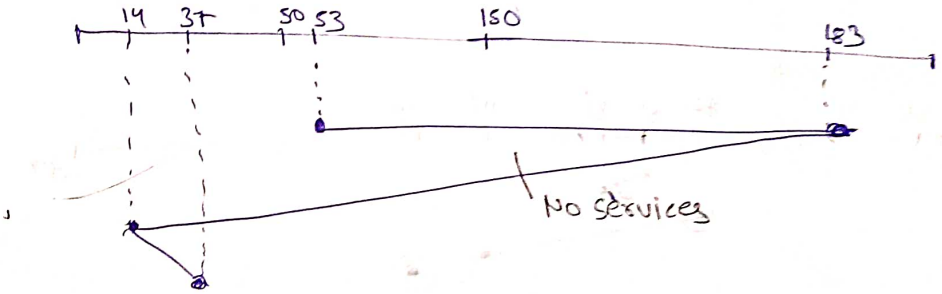
Seek time = (199-53) + (199-0) + (37-0)

= 382

6) C-Look

<Circular look>

* It doesn't go to last track (199) but goes to last req ~~track~~ track and similarly it doesn't start from the first track. ~~often~~



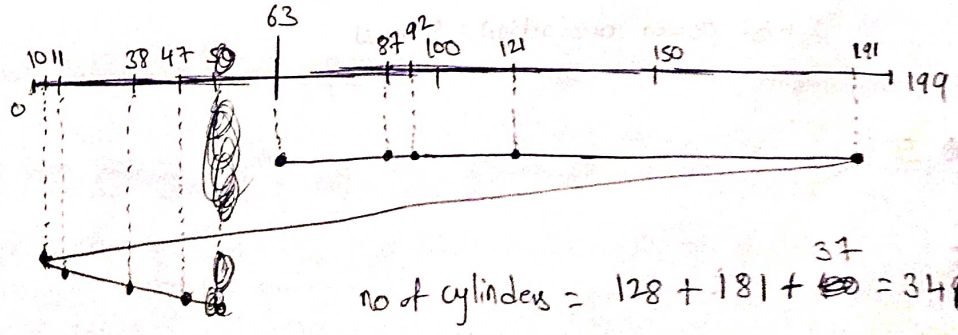
Seek time = (183-53) + (183-14) + (37-14)

= 130 + 169 + 24

Note :

* Generally SSTF is found to ~~be the optimal one~~ give more throughput. However it is not the optimal one.

H5/14



no of cylinders = 128 + 181 + 37 = 346

H5|15

FCFS

	1	2	3	4	5
Time of Decision	160	213 ms	286 ms	331 ms	
Pending requests	{1,2,3,4}	{2,3,4,5}	{3,4,5}		
head position	65	12	85		
Selected Request	$\{1\}_{12}$	$\{2\}_{85}$	$\{3\}_{40}$		
Seek-time	53 ms	73 ms	45 ms		

TTT = 1 ms

seek time = no of seeks * Track track time

Head = 65

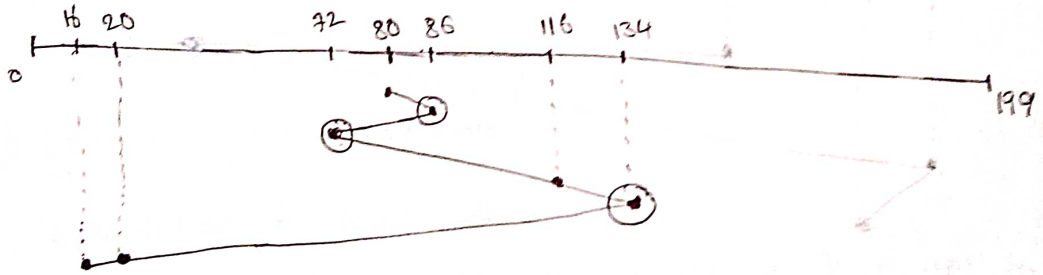
clock = 160 ms

21/08/20

H5|17

Reqs in terms of cylinders

<72; 134; 20; 86; 116; 16>



total seeks = 200

i.e., $\frac{200}{100} \times 20 = 40$ mw

no of R/w head reversal = 3

$\therefore 3 \times 15 = 45$ mw

$\therefore$ total power dissipation = 85 mw

H5|18 5

H5|19 10

(HS/20) Given is uniprogrammed processor.

Since it is uniprogrammed & Device queue will have only one process and any scheduling algo would give same performance.

∴ 0% improvement

(HS/21)

$$6ms = x(1) + (1-x)(10)$$

hit = $x = 4/9$

miss = $1-x = 5/9 = 0.55$

∴ miss ratio = 55.55%

∴ miss ratio < 55.55%

∴ 30MB cache

(HS/22) # N processes each of B bytes address space

∴ N * B bytes is the min size of disk space required

Leftover topics:

Inverted Paging (Reverse Paging):

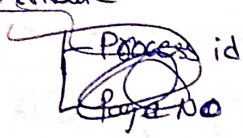
→ conventional paging designs page table over VAS.

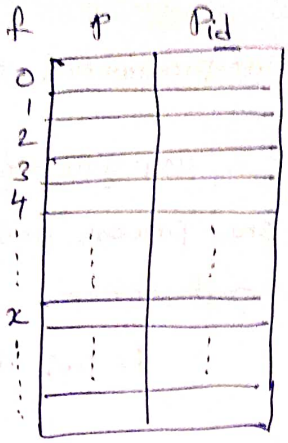
→ Inverted paging designs page table over PAS. We call this table as frame table. (Inverted page table)

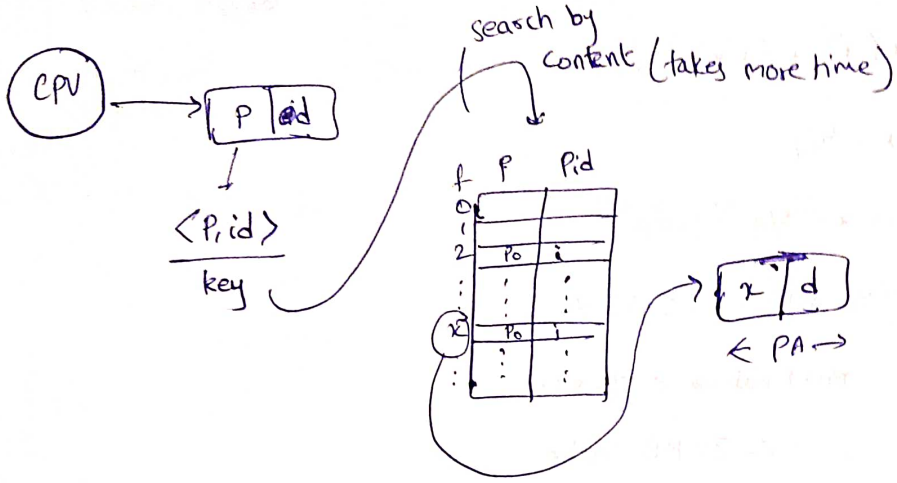
→ Frame table entries contain page no number ~~the core~~ that is ~~beg~~ being held by the corresponding frame. Also it contains process id of the corresponding process.

→ no of entries = no of frames

→ ~~two attribute~~







→ If no match is found for the search, a page fault is generated

Advantage:

→ One inverted PT for all processes (space efficiency)

Disadvantage:

→ More searching time.

Q33 VA: 34 bits PA = 29 bits

P.S = ~~8~~ 8 kB P.T.E : 32 bits (for both table)

Find size of PT & IPT

Sol:

Size of PT = no of pages * entry size

$= \frac{2^{34}}{2^{13}} * 4 = 2^{23} = 8$ MB (for one process)

Size of IPT = no of frames * entry size $= \frac{2^{29}}{2^{13}} * 4 = 2^{18} = 256$ kB

(for all processes)

Monitors:

→ High level language constructed supported by OS

→ Implemented as an ADT

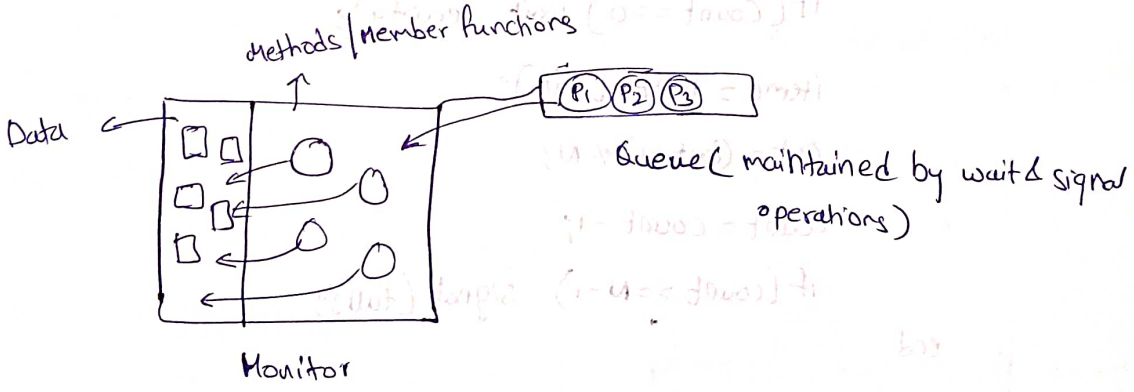
Defn:

Monitor is a collection of procedures, variables & data structures that are grouped together in a special package/module.

→ Procedure that runs outside the monitor cannot access monitor internal variables & data structures. However, they can activate monitor member functions.

→ Monitor include special condition variables that support wait & signal operations.

→ Monitors have an important property that only one process will be active in the monitor at any time.



→ A monitor, internally, has a mutex. Every process implicity performs down on the mutex before uses the monitor.

Producer-Consumer using monitor:

```
Monitor ProducerConsumer // creation of monitor
begin
    int count;
    Condition Empty, full; //condition variables
    Procedure Enter;
    begin
        if(count == N) wait(full);
        buffer[in] = itemp;
        in = (in+1) % N;
        count = count+1;
        if(count==1) signal(empty);
    end
    Procedure "Remove"
    begin
        if(count==0) wait(empty);
        itemc = buffer[out];
        out = (out+1) % N;
        count = count - 1;
        if(count==N-1) signal(full);
    end
    init: //constructor
        count=0;
end Monitor
```

→ Outsider the monitor:

```
Procedure Producer
begin
    while(true)
    {   produce_item(itemp);
    }   ProducerConsumer.enter
end
```

```
Procedure Consumer
begin
    while (true)
    {
        Producer Consumer. Remove;
        Consumer_item (item c);
    }
end
```

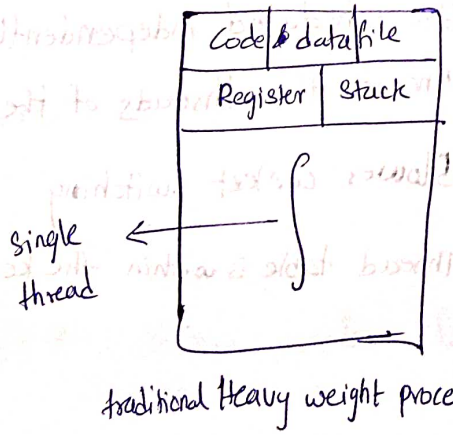
→ only one of the above two processes can access the monitor simultaneously and thus this soln is free from deadlock and it is consistent.

Threads & Multithreading

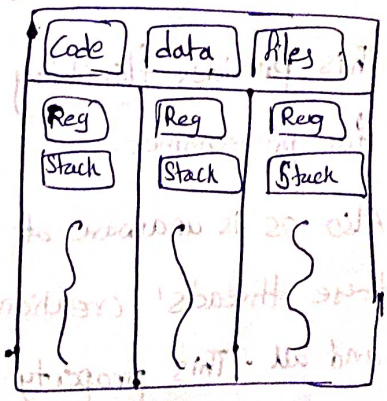
→ Thread is a light weight process

→ thread is an active entity and it is a unit of CPU utilization.

→ Schedulable unit.



traditional heavy weight process



* without threads we need to create multiple processes which share same code but still has multiple copies.

* Thus we use thread which has their own registers and stack but share common code, data & files.

→ Since threads share resources, we call it light weight.

Benefits of multithreading

→ Resource sharing

→ Cost effective

→ Economical

→ Less context switch overhead
(∵ size of TCB < size of PCB)

→ Achieves Computational speedup on multicore architectures.
(i.e., Parallelism)

global variables & heap are also shared

Every thread has a Thread Control Block

Types of threads

User level threads (ULTs)

→ Threads that are created & managed at user level is are called user level threads

→ This provides flexibility for programmer.

→ Also OS is unaware of these threads' creation and all. This property is called transparency.

Eg: Java threads. (managed by JVM),
Pthreads

→ Faster context switching (∵ no mode switching)

Kernel level Supported threads (KLTs)

→ Here we have multi threaded kernel.

→ ~~Thus kernel threaded~~

→ Thus kernel supported threads are scheduled independently be one of the threads of the kernel.

→ Slower context switching

→ Thread table is within the kernel.

→ One drawback of user level threads is, if one thread of process requires an I/o the whole process is blocked by the OS (∵ OS is unaware of threads in user level).
To overcome this disadvantage we go for kernel supported multithreading

→ ∴ user level threads need non-blocking system calls.

Ha/12

P(m);
P(m);
—— deadlock

∴ 1 process & 1 lock

∴ x=1 & y=1

→ Since threads share common address space communication blw threads is easy. (Eg: through global variables)
IPC blw processes is complex

→ Creating a process is costlier than creating a thread.

→ If a thread dies, its stack is reclaimed. If a process dies all threads die.

→ For user level threads, the user application maintaining thread table and kernel maintains process table

Threading issues

→ what if a thread invokes fork()?

- Duplicate all threads
- Duplicate only caller thread (easier to implement)

→ Segmentation fault in a thread

- terminate thread
- terminate whole process

* fork():

~~return value of fork.~~

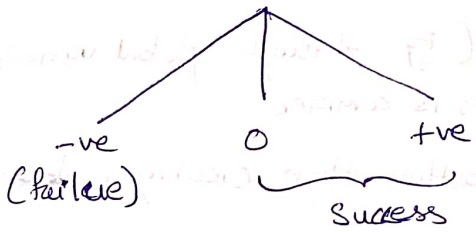
* If parent finishes execution before the child does, then such child processes are called orphan processes.

* Orphan processes are taken over by init process in Unix.

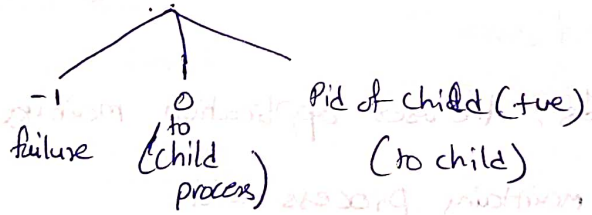
* If a child process wants to terminate but parent process is not waiting for them to terminate are called zombie processes.

  These ~~are~~ processes finish their execution but still has entry in process table

* Every system call has a return value which returned by the kernel.



* return value of fork()



* Parent & ~~child~~ child run parallely

Eg:
```
main()
{
    int id;
    id = fork();
    if (id<0)
    {   printf("error");
        exit();
    }
    if(id==0)
    {  //child process
    }
```

```
        else
        {   // parent
        }
    // child & parent
}
```

Eg:

```
main()
{
    int id, x=10;
    id=fork();
    x++;
    id=fork();
    if(id==0)
    {   x++;
        printf("%d",x)
    }
    else
    {
        --x;
        printf("%d",x);
    }
    x=x+5
    printf("%d",x);
}
```

o/p of parents:

10
15

o/p of child:

12
17

Eg 2:

```
main()
{   if(fork()==0)
        printf("*");
}
```

o/p: *

```
main()
{   fork();
    printf("*");
}
```

o/p: **

P/14 Let initial val of a=10

15, &a (of child)
(u) (v)

5, &a (of parent)
(x) (y)

$\therefore u = x + 10$

Address spaces of child & parent are different

$\therefore v \neq y$

$\therefore$ opt (b)

However virtual addresses of v & y could be same

H6/1 How many times '*' is printed?

we know that

for 'n' sequential fork() calls no of child processes created $= 2^n - 1$

$\therefore$ we have condition if (fork()==0)

only child executes it

$\therefore 2^n - 1$ '*'s are printed

H6/2 H6/4 else block corresponding to parent is accessing 'd'.

this generates a compilation error.

22/08/20

Process Management - II Material Questions

P/9 a) P(S) & V(S) are used to provide ME for process_arrived and process_left. The implementation & usage is correct

b)

P_a [0] P_l [0]

P_1 busy wait

P_a [1] P_l [0]

P_2 busy wait

P_a [2] P_l [0]

P_3

exists barrier P_a [3] P_l [1]

enters barrier again

P_3 P_a [4] P_l [1]

∴ Deadlock

P/10 from above observation opt (b)

P/12

P_1 : L [0]
while( ) ✗
Now L [1]
<CS>

P_2 : L [2]
while( ) ✓
L [2]

P_1 release lock
L=0 ; L [0] // lock available

P_2 : L=1 L [1] // CS is available but lock is not available

P/15 If context switching is disabled

imagine the process is busy waiting.

Now process will run for infinite amount of test.

∴ opt (a)

P/17 Clearly it does not guarantee ME

also deadlock free

P/18 for binary semaphore S

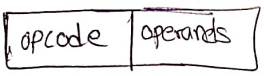
```
P(S);
P(S);
```

above code, cause deadlock

Memory Management Ace Material:

P/10 Here - min no of frames means the least possible no of frames required to execute one instruction.

Instruction format



x_1, x_2, x_3 (Assume indirect mode)

∴ no of frames req depends on the format of instruction and addressing mode.

P/16 I is true. But it is not done to prevent thrashing

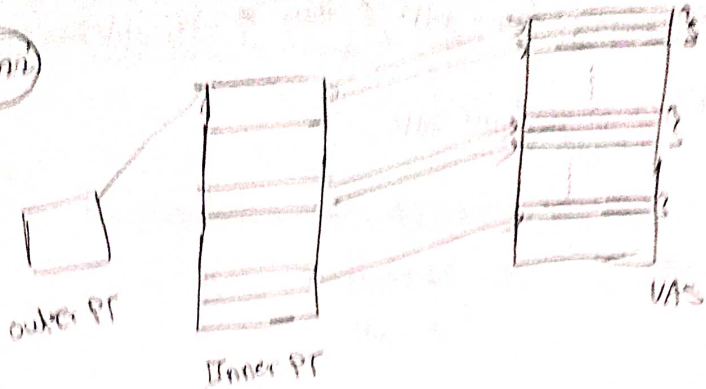
∴ only II is true.

P/21 $0.96[1 + 0.9(1) + 0.1(1+10)] + 0.04[\underbrace{1 + 2(10)}_{\text{for PA}} + 0.9(1) + 0.1(1+10)]$

$= 3.8$

$\therefore 4$ ns

P/22



No of entries in page table per page = 2^{10}

i.e., 1st page of page table has entries for pages 0 to $2^{10}-1$

0000000000 to

Base 0x0040000000 is greater than $2^{10}-1$

∴ diff pages of page table

Silly for 3 page

∴ 3 pages of Inner PT & 1 page of outer PT

∴ 4 pages = 4 × 4kB = 16kB

P/23

6 ⟶ 9

4 ⟶ 11

~~optimal has~~ 10

page faults in optimal ≤ page faults in LRU

∴ eliminate opt ⓒ

is ~~a~~

opt ⓐ shows belady's anomaly

∴ ad

opt ⓓ:

6 ⟶ 6 pg faults ⟹ there must be 6 unique pagRes

4 ⟶ 7 pgg faults

if we have 6 unique pages LRU with 6 frams should also give 6 page faults

∴ opt ⓑ

P/42 d & e surely reduces page fault rate & thus ↑ CPU utilization ✓

g & h may also reduce page fault rate

P/43

a) good

b) not good

c) good

d) not good

e) Pure Code means read only code

∴ we don't need to write back

∴ good

f) vector operations means array operations

∴ good

g) indirection mean linked list

∴ not good

P/46

Cycle time is memory access time

m ⟶ 1μs

other ⟶ (1+1)μs

Data transfer rate : 10^6 w/s

RPM : 3000

r rotational delay $= \frac{60}{3000} = \frac{1}{50}$

$= 0.02\,s$

$= 20\,ms$

rotational latency = 10ms

transfer time for 10^3 words (one page) $= \frac{10^3}{10^6} = 1\,ms$ — total : 11ms

EMAT = $0.99(1\mu s) + 0.01\left[0.8(2) + 0.2\left(0.5 + \underset{*10^3}{22} + 0.5 + \underset{*10^3}{11}\right)\right]$

= ~~$0.99 + 0.01[1.6 +$~~

P/47 when process goes to the block state

P/49 a) ~~logical address~~ · LA : 32

page no : 20

offset : 12

b) ~~fg no:~~

req info for translation is page number & corresponding frame number

$\therefore 20 + 12 = 32$ bits

$\therefore$ 4 bytes

$\Rightarrow \frac{64}{4} = 16$ entries

c) $\frac{2^{32}}{2^{12}} = 2^{20}$

i.e., 1M entries

P/54 a) P_0

b) P_2

c) P_1

d) P_0

P/56 a) $2(200) = 400$

b) $0.75(0+200) + 0.25(200+200)$

$= 250$

P/57 no of partitions $= \frac{2^{32}}{2^{20}} = 2^{12}$

$\therefore$ 12 bits

P/58 e, f

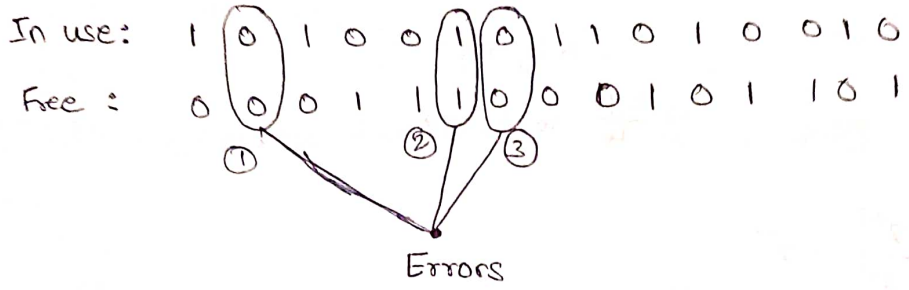
↳ due to belady's anomaly

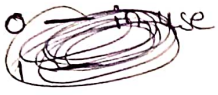
File System Material Questions

(P/9) **Spooling:**

when we need to print something, the CPU can't wait until printer takes up the printing job. So a spool is created on disk and O/P is kept there. A process spooler now prints all the O/Ps present in the spool. This concept is called spooling.

(P/11)

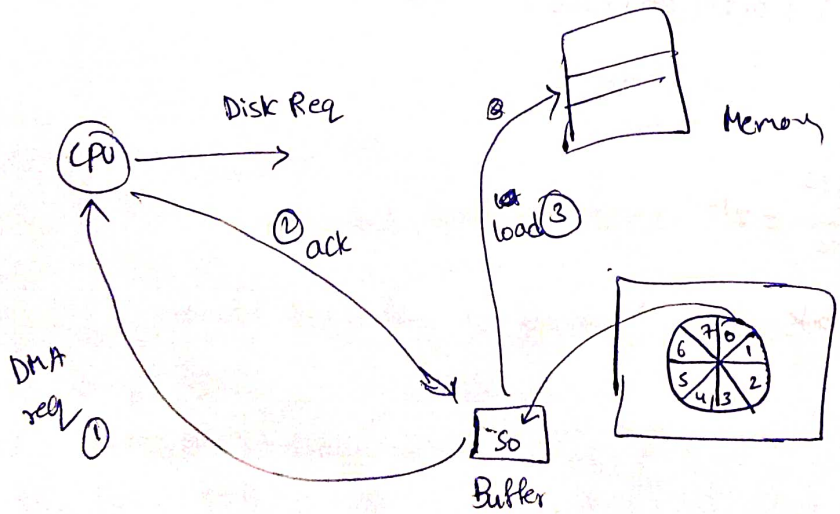




② is dangerous because it has a file and it also being show that it free. This causes data loss.

①&③ causes wastage of memory.
This type of wastage is called leakage.

Disk Interleaving:



Now in this case since ①②&③ may take more time

Say time req to load a sector into buffer.

In this way we ~~will lose sector~~ may not to have sectors 1, 3, 5, 7 loaded into the memory. even ~~after the~~ that they are loaded into buffer.

Now we have 2 solutions for this problem

(i) Hardware solution:

→ Here we maintain 2 buffers.

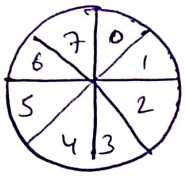
i.e., Double buffering technique.

However still if ① ② ③ (3 steps) take way more time than that of time req to load a sector we ~~may~~ cannot change no of buffers available dynamically.

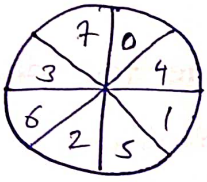
This the limitation of the solution.

~~(ii) Soft software~~

(ii) Interleaving (slw solution):



Non-Interleaved Disk (1:1)



Single Interleaved disk. (2:1)

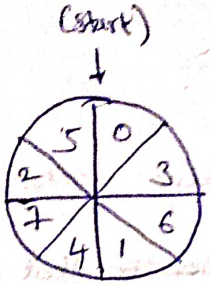
→ In ~~interleaved~~ single interleaved disk interleaving factor k is 1.

This single interleaved disk requires 2 rotations to access the complete track.

→ As k value increases no of rotations required increases.

↓

interleaving factor.



Double Interleaved disk ($k=2$) (3:1)

Here 1st rotation $\longrightarrow$ 0, 1, 2

2nd rotation $\longrightarrow$ 3, 4, 5

3rd rotation $\longrightarrow$ 6, 7 (0.75 rotation)

$\therefore$ total rotations required = 2.75

P|12

300 rotation —— 60 sec

1 rot —— $\frac{60}{300}$ = 0.2 sec = 200ms

$\frac{1}{2}$ rotation $\Rightarrow$ 100ms

$\therefore$ total time = 100ms + 2.75 * 200ms

= 650ms

~~Data rate for dou~~

for non-interleaved,

$$\text{data rate} = \frac{8 \times 2^9}{200\,ms} = \frac{8 \times 512}{200} \times 10^{-3} \text{ Byts/sec}$$

for double interleaved,

$$\text{data rate} = \frac{8 \times 2^9}{2.75 \times 200} \times 10^{-3} \text{ byts/sec}$$

$\therefore$ Data rate is reduced by the factor of 2.75

P/13 a)

no of addresses per block = $\frac{2^{13}}{2^2} = 2^{11}$

max file size = $12 \times 8 + 2^{11} \times 8 + 2^{22} \times 8 + 2^{33} \times 8$

$\cong 2^{46}$ Bytes

$= 64$ Terabytes

However disk capacity = $2^{32} \times 2^{13}$

$= 2^{45}$

$= 32$ TB

However for this question

Ans: 64 TB (∵ it is asked for this system)

b) 8 bit pointer $\Rightarrow 2^8 = 256$ partitions

no of blocks per partitions

$= 2^{24} \times 8$ KB

$= 2^{37}$

$= 128$ GB

Note:

→ To increase efficiency most of the file systems groups blocks into larger chuncks called clusters.

Disk Ilo is done via blocks.

File system Ilo is done via clusters. (assuming that file system Ilo has more sequential access character.)