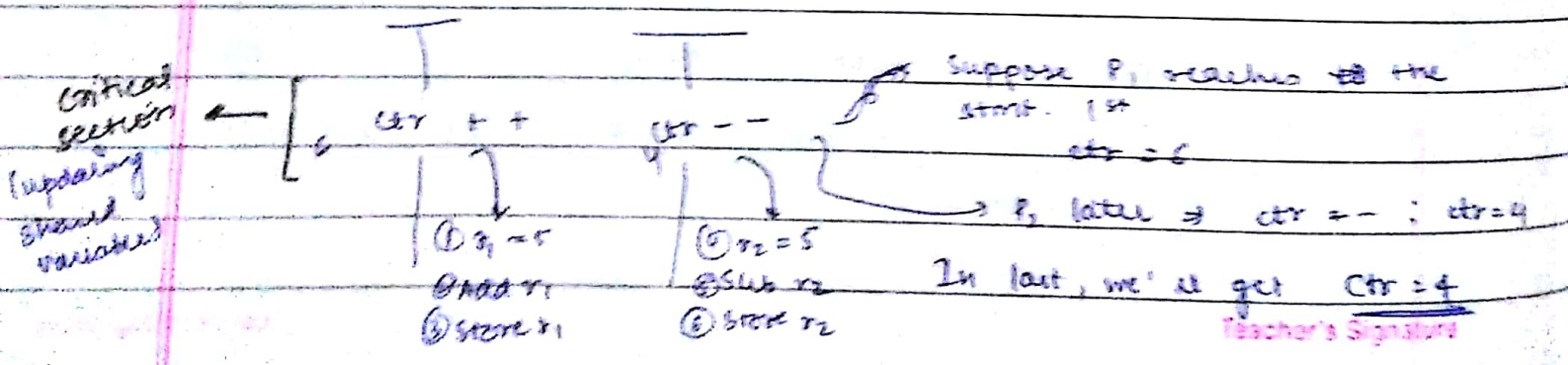
PROCESS SYNCHRONIZATION

→ multiple processes running I we end up getting an undesirable result

↓
can be dependent / independent

→ Synchronisation : Synchronizing program to get desirable result

Eg: Suppose P_1 & P_2 are running concurrently (Both using common variable ctr)



Teacher's Signature

* Before ② & ⑥, if ①, ③, ⑤ & ⑦ are executing → they'll take ctr = 5 only in both processes P_1 & P_2.

Problem : Upgraded value is not reflected to other party

↳ If single inst^n is executed ⇒ then P_2 will take ctr = 6 & execute further

↳ But generally, inst^n are executed multiple at same time.

→ To avoid this, we can execute programs one by one

02/18

Critical section Problem : above problem is known as

↓

section where updation of shared variable takes place

→ won't execute in parallel

To avoid: If one process goes into critical section, other 'll go only after 1st process has come out of that section ⇒ Soln to problem of Critical section

→ The soln should satisfy some properties :

→

① Lock : If one gets lock, other should wait until 1st has unlocked.

1) Mutual Exclusion

2) Progress

3) Bounded Wait

1) Mutual Exclusion

Only 1 of them is allowed to enter in critical section.

Teacher's Signature

2) Progress :
within any time limit. they should make some progress.
(Both don't go to wait state).

3) Bounded wait :
If one requests to enter in critical section & other has got entry. Bound wait tells how much 1st has to wait. It (both in infinite loop) shouldn't be in the situation that 1 keeps getting entry and other just waits. Each process will get chance when its limit is over.

→ assumed that they are performed in infinite loop

```
while (true)
{
    Entry Section      → process requests for lock here.
                         If gets : proceeds to critical
                         section
    Critical section
    Exit Section       → releases its lock
}
```

First sol^n to this problem :

Peterson's sol^n :

Used 2 variables :

int turn — 0 (P0) / 1 (P1) (to represent 2 processes)

boolean flag[2]; (T/F) (to show interest of process for entry in critical section if interested → Tanker's Signal)

P_i	P_j
while (true)	while (true)
Flag [i] = True	Flag [j] = T
turn = j	turn = i
while (turn == j && Flag [j] == T)	while (turn == i && Flag [i] = T)
;	;
critical section (CS)	critical section (CS)
Flag [i] = F	Flag [j] = F

(shows interest into entering — "मैं तैयार हूँ"; other may proceed if interested (manners))

interest of P_i

wait here

(need to check continuously & busy waiting unlike sleep(6))

suppose for both flag is true

Assuming turn = j is executed 1st ⇒ turn = j → P_i will wait

Goes to P_j → turn = i , while loop of P_i → false ⇒ It'll execute P_i का CS

Flag [i] = F ⇒ while loop of P_j ⇒ f ⇒ wait is over

↳ It satisfies all 3 properties mentioned earlier

→ This is a hardware level solⁿ.

→ s/w level solⁿ : (in pthread library)

<u>Mutex Lock</u>

↓

→ boolean available

→ 2 func's : acquire / release lock

If available ⇒ acquire it

Otherwise, wait

→ acquire funcⁿ : Mutex Lock

release funcⁿ : Mutex Unlock

Teacher's Signature

Code:

In func':

for loop : just for waiting

counter : shared variable (global)

goes into critical section after allowing lock

↓

pthread_mutex_lock(&lock);

* 1st use mutex_init() to → gives non-0 value if success
initialize default value

→ Once a process has acquired lock, other '2l just wait

* mutex can only be used in case of 2 threads

* In case of multiple threads : 'semafour' technique → implemented with help of integer

↓ ↓

Binary Counting

2 values : 0, 1) ↓ tells

↳ Alternate things ↳ Can synchronize 2-t. only 1 executes at a time

Every turn, how many times a thread will be executed

1 → gets entry (available)

Else gets value 1 & all other 0

Code:

wait : wait till gets lock

↓ ↓

if 0 if 1

Sem_wait will decrement the value

proceeds → gives access & again decrement value of progress to 0

Teacher's Signature

tells whether variable is shared or not

sem_init (&ping_sem, 0, 1) → semaphore variable

→ sem_post : to allow other threads to go ahead

ping
pong
ping
pong

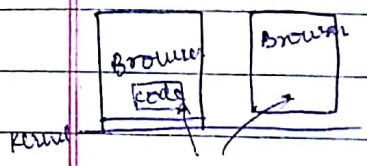
→ sem_init (, 0, 2) → even if increment once, '1' is still go for same value

ping
ping
pong
pong

→ adds (0, 1)
(0, 0) adds semaphore value (0+1) here

→ (, 0, >1) → Counting semafor
(na 0,1)

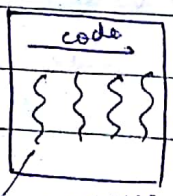
5/3/18



If Browser needs to interact with hw, it 'll give system call

Some part looks for user requests.
If 1 more tab needs to be opened
it 'll fork();
some memory will be added (more space)
exactly same code is running
use
extra allocation of Resources. (not needed)

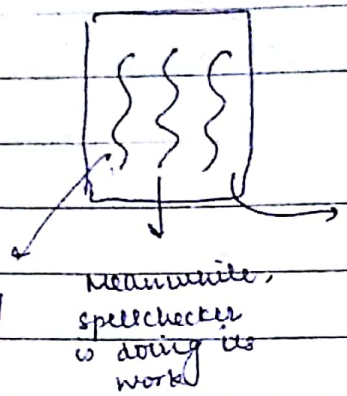
If running same code, we don't need to start new process (fork)
In this process only, we can have parallel execution flow of control



haven't created separate process.
Using thread, not fork
creating new process.

Teacher's Signature

→ Text editor : Multi-threading



* At User level only, threads are created. OS doesn't know about it. It isn't designed in a way to recognize threads.

→ When Browser is scheduled, one of its thread will get CPU.

→ However, when Kernel is doing threading, it doesn't see 1 process as 1 entity. So, multiprocessing can happen. All threads of a single process can run simultaneously. If multicore.

Deadlock

There are limited resources (for which each process fights) in a system

2 types

Physical	Logical	* Not Virtual add. space because each process has its own virtual add. so they don't have to fight for it.
RAM, ~~Cache~~,	Files,	
CPU, secondary storage,	deadlocks,	
I/O devices	Semafors	

↳ Deadlock :- ~~One person has~~ Each process is holding one resource & waiting for other resources which are held by other processes. ~~Have set of active processes~~ atleast n

using system call

→ Everytime a process sends a request for resource, OS keeps a system table having info. about each resource & which process wants it.

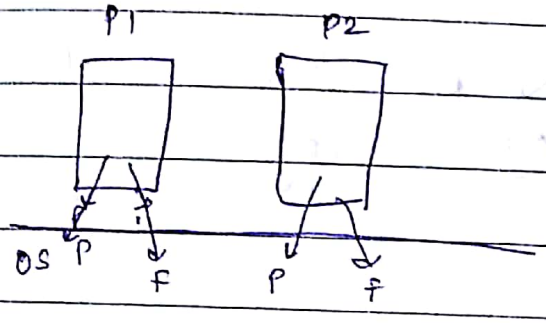
Process may have to wait in a queue.
↓
may release the resource later when finished.

3 things

1.) Request :

2) Use :

3) Release :



Active processes

P1 : wants to use ~~disk~~ File & Printer

→ sends request to OS for Printer & file

P2 also sends " "

P1 gets P & P2 gets F

→ system is deadlock

→ OS checked system table, saw P is free & assigned P to P1 & made note that P is assigned to P1. P2 got scheduled after this. It checked P is already assigned, F is free, so assigned F to P2. Again P1 scheduled, F is not free, P1 is put in queue for F → Deadlock.

P2 " " " P

If 2 printers : 1 with laser, 1 with not
⇓
They're different resources

DATE: / /
PAGE NO.:

4 necessary conditions for deadlock to occur :

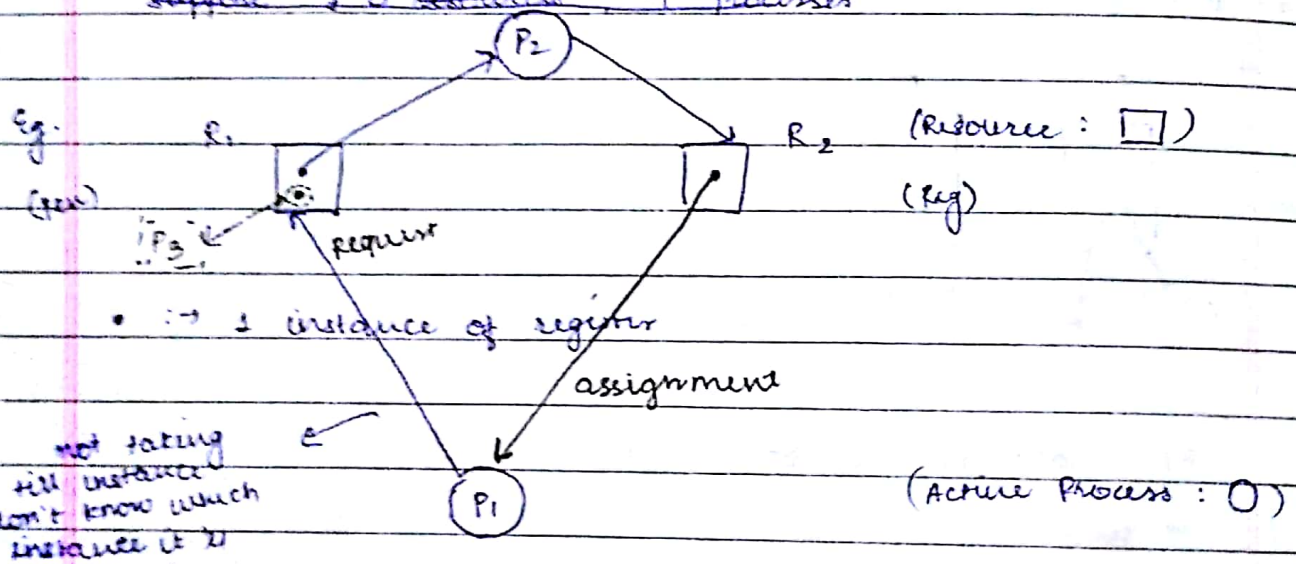
1) Mutual Exclusion : Resources are non-sharable
(If 1 is using, other won't be able to use it)

2) Hold & wait : Each process is atleast holding 1 resource & waiting for other resources.
↳ involved in deadlock.

3) No Preemption : No one is releasing the resources once it has held it

4) Circular wait : Each one is waiting for res. held by next & last one " " " " " " first

~~Suppose 2 S resources 4 Processes~~



Eg.

• :→ 1 instance of register

Resource allocation graph

P1 has register & waiting for pen

→ RAG : has vertices (V) & edges (E) (directed)
- R2 → P1 : (assignment)
- R1 → P2
- P1 → R1
- P2 → R2 : (request)

Processes (P1, P2, ..., Pn)

Resources (R1, R2, ..., Rn)

** there is a cycle in a graph ⇒ a deadlock may exist.

If there was another instance of R1 available & assigned to P3

there's still a cycle but deadlock won't happen. When P3 finishes that instance will be assigned to P1. So now, P1 can execute & then release all its resources. P2 can run now.

9/3/19

→ How to handle deadlocks?

Handling deadlocks

Prevent, avoid | Detect & Recover | Ignore

① Prevent - don't allow any one of the 4 cond"

To prevent :-

↳ Mutual Exclusion :- Everybody can share resources. This isn't possible. Whatever mutually exclusive resources are + nt in system, they'll remain " forever. So, this can't be changed

↳ Hold & Wait :- If you're requesting for another resource which isn't available, leave your own resource (release). Don't hold & wait.

I'm continuously requesting for resources but not holding anyone.

↳ No preemption :- depends on implementation of OS, priority of process. OS can preempt resources. It's possible

↳ Circular Wait :- Assign some no. to all the resources.

R_1 — 1

R_2 — 2

R_3 — 3

R_4 — 4

If I need R_1 & R_3, we can get the ordering, i.e., first I need to have 1, then only I can have 3

got

P_1 — 1, 2

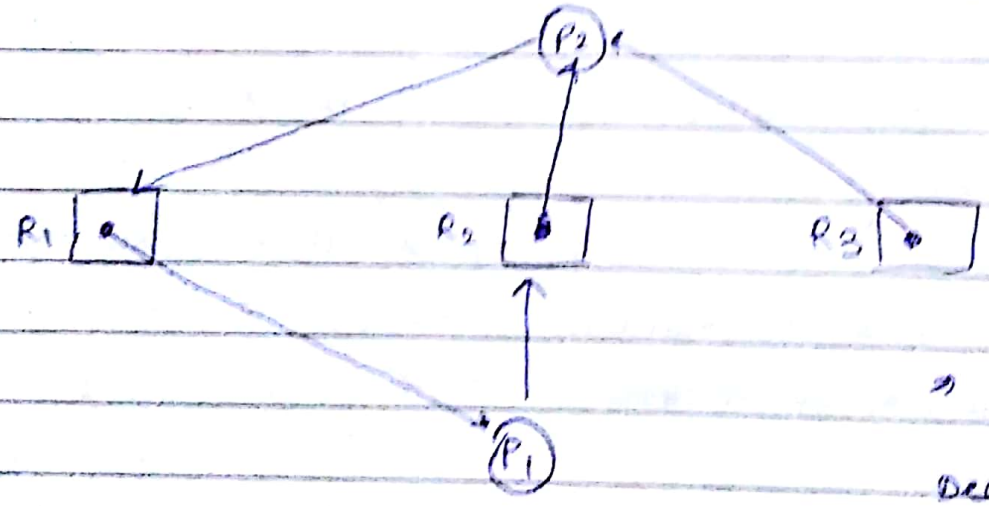
P_2 — 1, 3, 2 (Ordering has to be within the resources which it wants)

Suppose no ordering

P_1 got → 1

P_2 got → 3

] ⇒ deadlock may occur



⇒ cycle ⇒ Deadlock

If have ordering

P_1 got → 1

P_2 ——→ wants 1, won't get it

it won't get 3 because it didn't get 1

can apply for R_j only when I've R_i, $i < j$

2 resources of same type → 2 instances of R_1.
(R_1)

DATE: / /
PAGE NO.:

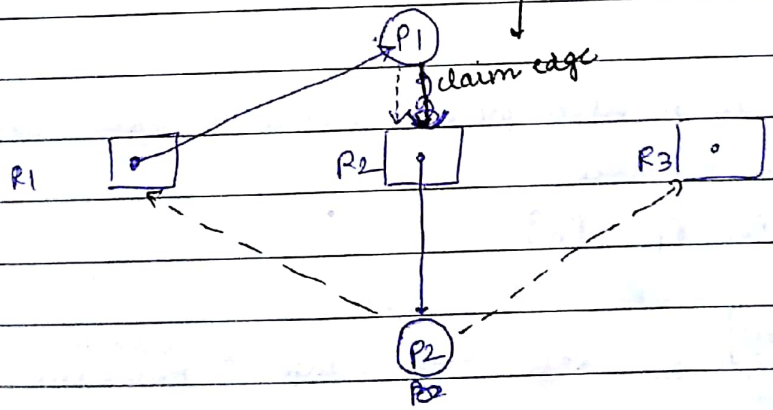
→ P_1 got 1
P_2 got 2
) → again a deadlock if no ordering

Avoidance: can happen only when we've prior info. about what resources are needed by a process in a long run.
specifies everything in the beginning
* OS

Demand made to OS (Resources needed)	≤	Total no. of resources of system (max. resources)

Suppose P3 has been allocated resources initially. Now, P1 comes & requests for some resources. If I've them, give to P1. Now P2 comes & " " . If I don't have them, P2 has to wait.

↳ Suppose P1 announced earlier it might require R_2
P2 " " " " R_1, R_3



Now, P1 is requesting for R2
claim edge ⟹ (converts) request edge
Algo checks if claim edge → " , is there a cycle or not. There is a chance of cycle so it won't be given request change. (depends on your system & how many claim edges you may have)

Scanned by CamScanner

here we r not allowing P_i requires R_2. If this happens

wait & wait have occured.

→ Now, if P_1 requests for R_3, then safe cond^n will be raised because initially, it didn't give the need for R_3

* If cycle may happen ⇒ avoid it
→ not very efficient for multiple instances of resources.

Bankers Algorithm :-

Bank - taking care of resources just like OS

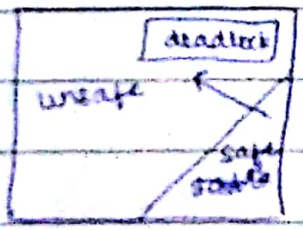
	Max needs	Current Needs	Free: 3 More Needed
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

× × ×

* Safe state :-

If a sequence in which processes can be allocated resources, it is a safe sequence.

eg $P = \{P_1, P_2, \dots P_i, \dots P_n\}$



safe ⟶ unsafe ⇒ deadlock may occur

if P_i is allocated some resources & it needs some resources which are held by P_j, $j < i$ ⇒ it is in safe state.

sequence of process execution so that resources required are fulfilled by the resources it is currently holding + resources that would be released by other process.

→ If P_i has 3, requires 10. Other 7 are held by 3 processes before, it has to wait for them to occur.

eg. 3 12 printers in system

		max	currently,
P_0	→	10	allocated : 5
P_1	→	4	: 2
P_2	→	9	: 2

9 have already been allocated

→ Free : 3 printers

We want to find a safe sequence.

P_2 only needs 2 more ⇒ first start with P_1 & allocate 2 to it

Free : 5

P_0	10	5	5 →
P_1	4	2 →	2
P_2	9	2	7

$< P_1, P_0, P_2 >$: Safe sequence.

(If I run in this manner, deadlock will never occur)

When P_1 finishes, 4 more printers will be free ⇓ total : 5

Now, P_0 needs 5 more ⇒ it'll execute next

after P_0 execution :

Now, free = 10

Now P_2 will execute.

Teacher's Signature

→ request

use

release

* This algo starts from starving

→ If total printer were 10

free : 1

OS knows ki free one honge lekin nhi ki ek bhe process chal paayega ya nhi , toh no current needed bcoz same allocate nhi kr rahega

It check if safe sequence is possible after every allocation

9/3/18

→ Available [m]

Allocated [n x m]

→ Allocated [n][m]

→ Max [n][m]

(m types of resources) R1 R2 R3 R4

[7 6 3 4]

⇓

6 instances of R2 are available right now in system

n processes, each can have max. m resources

	R0	R1	R2
P0	5	6	2
P1	3	0	0
P2	1	8	3

P0 is allocated 2 resources of R2

↑

Max would also be like this.

⇓

gives the no. processor has submitted how many resources it might need during complete execution (process gives info. at the beginning)

Teacher's Signature

Ex.

	Allocated			Maximum			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	10	5	7
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			
	7								

Available after Allocation :

A	B	C
3	3	2

might request

Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

* If a process is requesting for a resource, it should get that resource.

→ Finish = [0 0 0 0 0]

⇓ gives which process has finished

Start from P_0, see whether :

1.) Finish [i] = 0 ✓

2) Need is less than Available resource ✗

⇓ can't give resources to it

go to P_1 ✓ satisfies need < available

P_1 will execute and finish

→ F = [0 1 0 0 0]

Now, whatever P_1 was having will be released.

Available : 5 3 2

Teacher's Signature

P_1, P_3, P_4, P_0, P_2 > : safe sequence

DATE: / /
PAGE NO.:

→ Now go to P_3 (next index) ∵

→ P_3 ✓

F : [0 | 0 | 0]

Available : 7 4 3 (add 2 1 1)

→ P_4 ✓

F : [0 | 0 | 1 | 1]

Available : 7 4 5

→ P_0 ✓

F : [1 1 0 1 1]

Available : 7 5 5

→ P_2 ✓ [1 1 1 1 1] : Once we get all 1, it'll terminate

→ This is how it is checking the safety.

→ When these algo run at regular interval, they cause overhead to the system.
That's why. OS chooses to ignore / sleep.

→ How system knows there is a deadlock?
They check CPU utilization at some interval
↓
if ↓ than general limit ⇒ some problem

Request :-

* If somewhere in b/w, $P_1 \rightarrow 1, 0, 2$
request comes in b/w by P_1

→ Check 1st : whether request < need
If not ⇒ It'll be error
(It wasn't requested in beginning)

Teacher's Signature

If yes, start allocn again.

Assume I've given ~~res~~ these resources to P_1.

After execution, available : 5 3 2

Again check for safe sequence.

↳ we will let P_3 to execute first

*) If P_3 asks for resources in b/w, it depends on no of resources requested whether we'll get safe sequence or not.

→ will check safety after ~~every~~ each interval.

* If it is in Ready Queue ⇒ of Printer ⇐ it has requested for Printer
(not in " of CPU)

→ If P_2 has priority over P_4 every time P_2 is being executed P_4 is always in wait Queue ⇒ Starvation

(Need > Available) ⇒ may lead to Starvation

↳ or give some resources & wait for some. ⇒ Hold & Wait condⁿ may arise deadlock
(1,2) (3)

↳ suppose

P_3 → 1, 0, 2

Assume that I've given & then compute safe sequence.

If no safe seq. ⇒ I shouldn't accept the request. → lead to unsafe state

If safe seq ⇒ I can consider this request

→ Banker's Algorithm → solve eg.

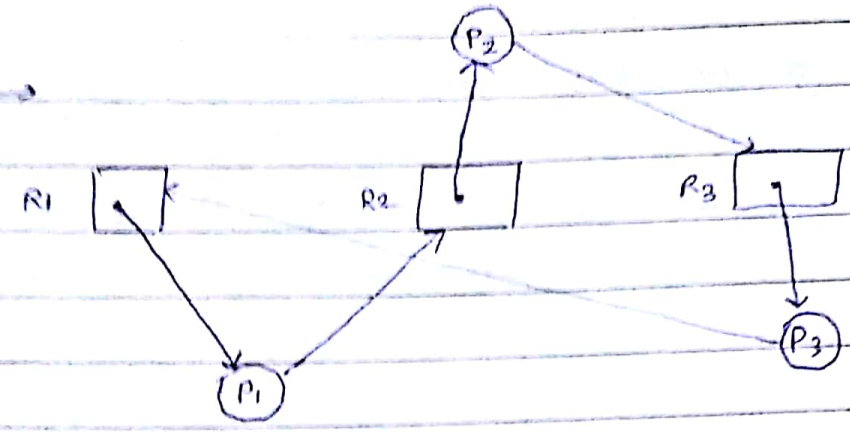
Deadlock has occurred we need to detect it

② **Detect** : (no prevention / avoidance)

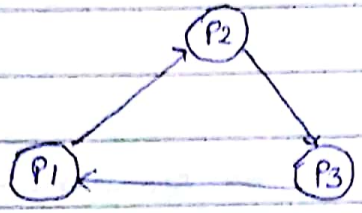
If single instance : Wait for Graph

Prevention : →

(Resource Alloc" Graph)



Detection : Wait for Graph (Remove resources)



⇒ Deadlock has taken place (cycle)

Wait For Graph

Recovery :

Can be done using Processes & Resources

1) Abort / Terminate processes : figure which process needs to be determined → depends on how far process is completed which is going to cause the min^m rollback cost.

Termination should also bring system from

2) pre empt Resources → figure out which process resource should be pre empted.

12/3/18

→ last eg. : for request case :

$Request_{P_1} = (1,0,2)$

① 1st check if Request < Max

↓ Yes

Assume I've given resources it to it

	Allo			Need			Available		
P1	3	0	2	0	2	0	2	3	0

→ until & unless system don't know the system is safe, it won't give resources to P1. It first assumes

P1 ✓ [0 2 0]

→ find a sequence so on

⇒ If $Request_4 = (3,3,0)$ (after above)

P4's request get rejected

	Available			Need		
P_0	0	0	0	7	4	3
P_1				0	2	0
P_2				6	0	0
P_3				0	1	1
P_4	3	3	2	1	0	1

Not possible after this assignment

* Request ko baad everytime I start from beginning

Teacher's Signature

If I can get $< P_1, P_3, P_2, P_4 >$

but can't execute P_0 still ⇒ Request can't be granted.

⇒ System will remain in its original state.

P_4 will have to wait in queue for particular device.

meanwhile, if some process gets finished & releases its resources, P_4 may get executed.

→ If can't get a safe sequence ⇒ it may lead to deadlock

→ If simultaneous requests come ⇒ it 'll depend on Queue that which request is in front of queue

VIRTUAL MEMORY

→ 32-bit processor : Range : 0 - $(2^{32}-1)$ address

→ 32-bit machine ⇒ word size ⇒ 4 Bytes

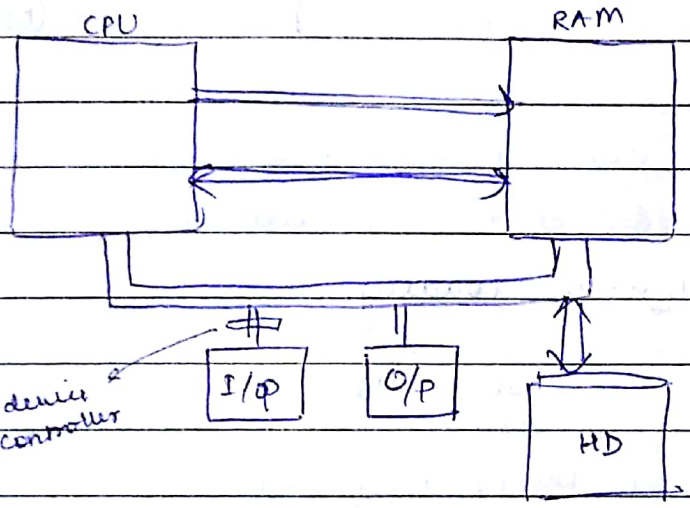
→ 64-bit machine ⇒ word size ⇒ 8 Bytes

→ 16-bit machine → ⇒ 2 Bytes

Byte Addressable ↑ because :

↳ saferg space

↳ most of data types are 2 byte, 1 byte ...
more accessible



Out of Jobpool (I/p queue)

↓ Longterm scheduler (program in OS)

picks program & put in RAM.

scheduler Healthy mix of I/O bound & CPU bound process

↓

which takes no time in computation

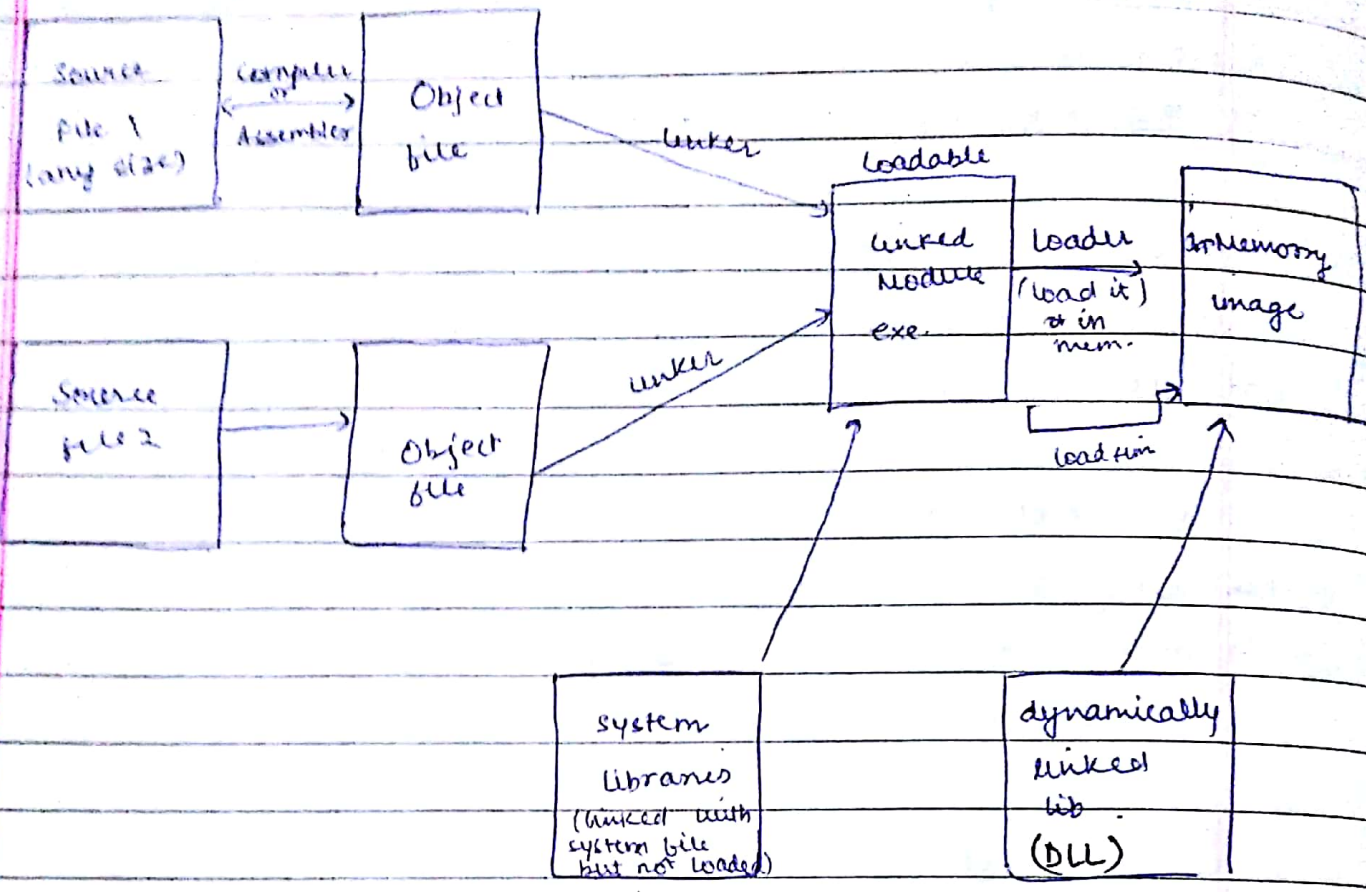
very interactive

14-09-18

→ If all i/o bound : CPU is idle, prog. all in queue for I/O

→ If all CPU bound : all in ready queue → WT ↑ TT ↑ waiting for scheduler

Teacher's Signature



- When loading in memory, there may be many routines (modules) which aren't needed at time of execution.

So, we use Dynamic Loading

↓

Routines / lib. will be loaded only if required. Otherwise, kept on backup storage (HD)

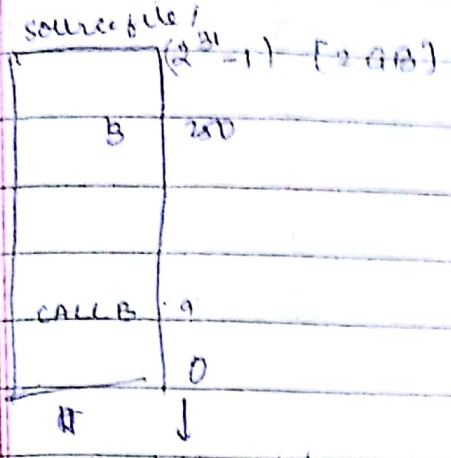
→ Dynamically linked lib. : small piece of code → stub (tells in module)

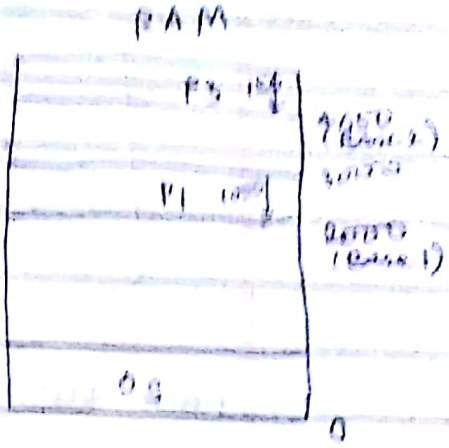
If this lib. is to be called, then how will it be called.

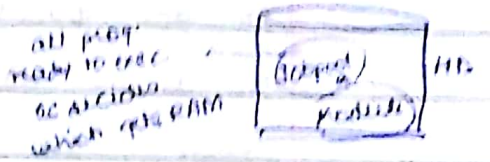
- ~~load~~ linked with prog. only when needed. Not initially linked.

→ ILC : Increment by length of last inst^n.

2 - pass Assembler







Where actually int in RAM : Physical Add

→ Add. binding : Actually putting phy. add. in process. Maybe at

(1) Compile time : We know where prog will be loaded in RAM at compile time only

Relocation const. : 2000 (let say prog. is loaded at add. 2000)

⇓

shift everything by 2000

add. binding has taken place at compile time.

Usually, this doesn't happen.

(2) Load Time : If OS gives info. when it is being loaded.

So, I know relocation const.

(3) Execution Time (Run Time) : know only at runtime.

→ Talking about RAM : (above diagram)

P1 & P3 loaded.

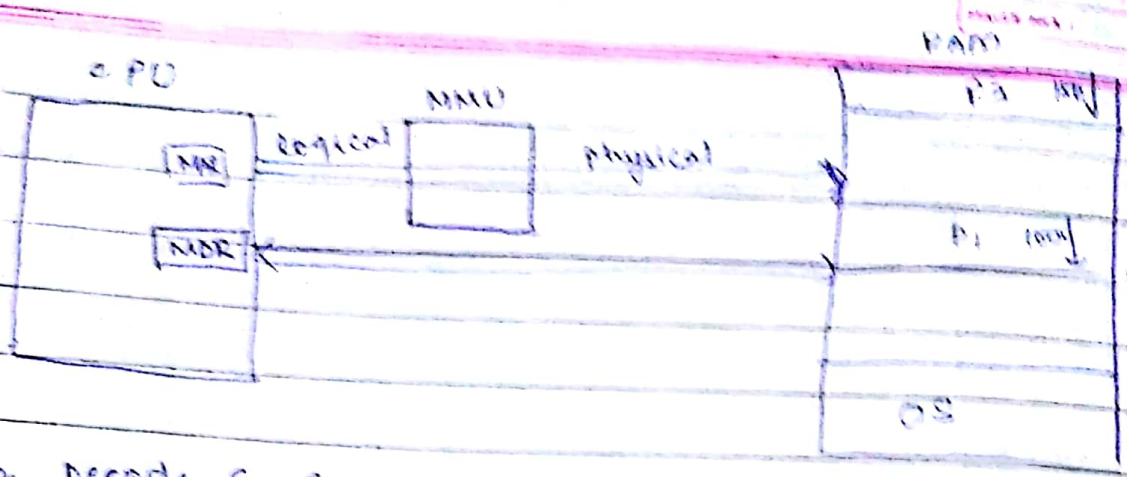
OS decides where they would be put

For P1 → 1000 : limit

P2 → 150 Bytes ; limit

Teacher's Signature

→



- Fetch, Decode & Execute.

- If add binding took place at compile time → P1 will be loaded at 2000 every time because put at compile time : can't be changed Same is problem with load time Usually, we don't prefer it.
  If 2000 isn't free → P1 can't be put (Hardbounded add.)
  OS don't prefers this

→ add. binding takes place at run time
P1 total size : 2 GB
when CPU generates add., it generate logical add.
only (+nt in inst^n)
P1 is generating all 2GB add.

→ MMU : logical $\xrightarrow{\text{mapping}}$ physical

Suppose, prog. is malicious & want to access other user's prog.

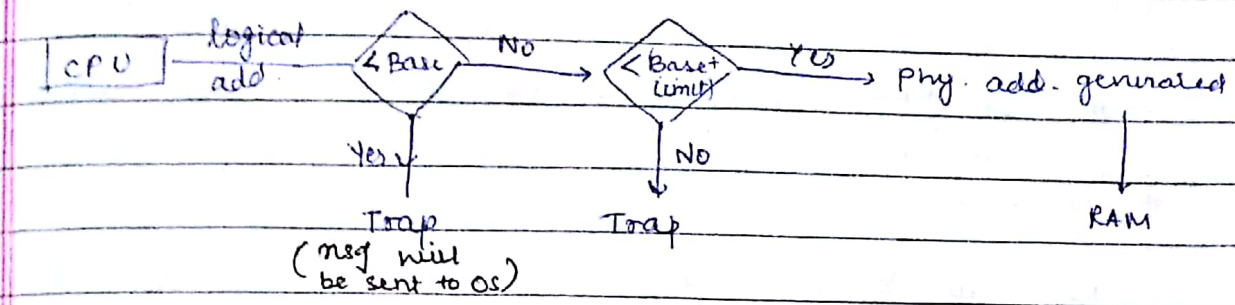
In P1, write JUMP 4000
⇓
Through mapping, can go to there. But this isn't allowed

No process can access other process's add. space (security issue)

Logical add. < Base (should be)

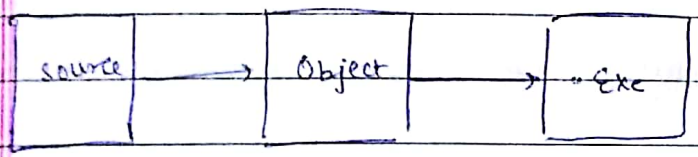
DATE : / /
PAGE NO. :

→ This should be checked at MMU to restrict this.
(2000 has been added to all inst^{n}s already)



→ Base & limit are stored in some reg. for each process

16/3/18



:→ Ultimately going to be stored in HD

Image (.exe)

ILC			
0	ADD R1, R2	2000	201 - - -
4	MUL - - -	2004	- - - -
8	CMP . . .		
12	JMP B	2012	- - - - - - 2100
100	B : SUB	100	- - - - - -

→ after shift
→ binary for 100

Assumay All ins^{n} → 4B

Metadata (info. about Object module & reloc^{n} const.)

→ At Compile time

→ Suppose, at compile time only, I know this prog. will be loaded at 2000 only. → I can already shift all add. by 2000 at time of compilation only.

→ Everything is hard coded ☺ at compile time only

→ There's no flexibility for OS now. It always have to be loaded at same place in RAM.

Add Binding → 2100 is already in program loaded

→ At load time : OS adds 2000 at time of loading

→ again hardcoded in Image when CPU sends add,
this is time it checks if
base < add < base+lim

→ At run time : Even image has JUMP 100, prog loaded
in RAM still has JUMP 100

Now, at exec. time, 100 will be sent from to CPU.
It'll check it is within limit or not
Yes ⇒ MMU will take care
↓
has relocⁿ reg. add it to 100 & thus,
phy. add will be generated

→ gives OS flexibility of putting the program anywhere in RAM

→ ~~Suppose 5 processes are in Jobpool~~

→ Backing Store : All processes in Ready Q are kept here
small, fast HD : can be in main HD or separate
OS may use it or not (associated with RAM)
↳ either put in RAM and give CPU 1 by1.
suppose high pr. process come in b/w & no space is available we take 1 Process & put in Backing Stor. Many OS avoid using it due to Overhead

A prog. can be a
→ given max memory
→ brought back to Backing stor
→ swapped ⇒ 1) lot of Overhead (Time ↑)
2) If I/O interacⁿ : I/O buffer used
↓
~~somewhere in RAM~~

suppose we swap P1 with P2 ⇒ need to make sure there is no i/o performed.

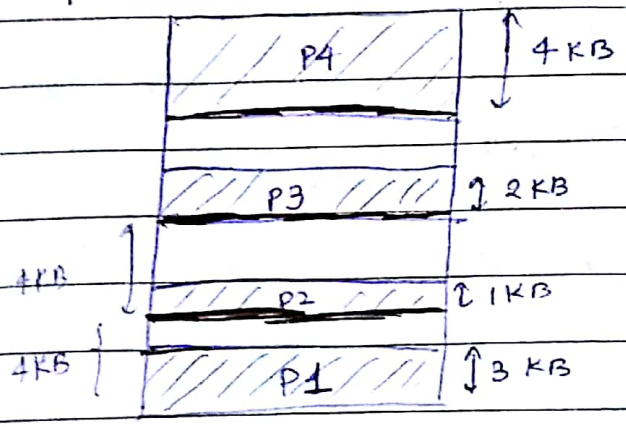
→ If swapping : better to use OS memory for i/o buffer. ⇒ P1 ke liye buffer abhi bhi available hoga. P2 can access it.

→ Backing store is same as swapping area.

→ Jobpool : has Processes which are ready to execute. Some of them are picked by long term scheduler & put in Ready Q

→ 2 ways of loading RAM programs in RAM : (Contiguous Mem. Alloc^n)
whole process needs to be put together

1) Multiple Partition scheme : 4 KB - partions : 4



OS mem (ut)

→ Each time bring a process, ~~put~~ ~~it~~ ~~it~~ give it a part^n.

→ Now, another process needs to be ~~to~~ brought in of size 6KB (If came earlier, can give 2 partitions to it)
Now, can't ~~give~~ bring it because no contiguous mem. space are available.

→ This leads to Internal fragmentation
wastage of mem.
(Each block is internally fragmented to holes)

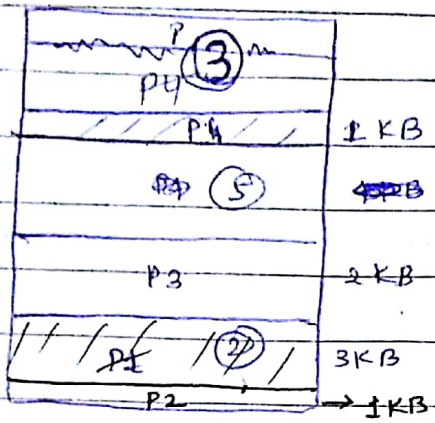
→ Even though mem. space is available, ~~it~~ process can't be brought in RAM
↓
can't use it

→ Wastage of Memory

→ Couldn't do anything to avoid holes.

2) Variable Partition Scheme :

not defining block as such .



whatever P1 needs, give it.

suppose P1 had to go for some I/O, ⇒ gone out of Ready Q

→ swapped out P1

suppose P2 came ⇒ hole

P4 swapped ⇒ hole

holes created outside add space of process

→ No block system here

→ External fragmentation is taking place

(External to process : whatever fragmentation is taking place)

→ again can't allocate 10KB program mem.

→ If N allocated blocks ⇒ 50% rule

↓

0.5 N blocks are fragmented (not used)

→ could can do : Compaction

move all processes towards 1 side & holes to another

OS keeps track of holes & processes

Suppose P6 comes → 2KB size

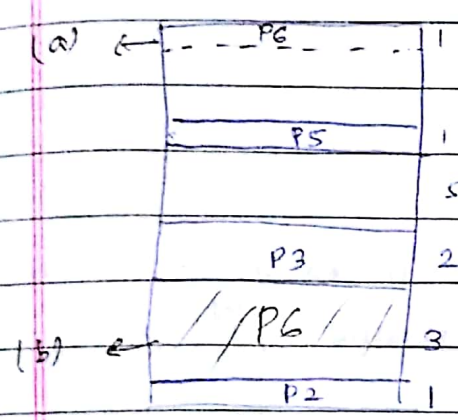
algorithms to put it :

a) first fit : scan list (from top / bottom)

1st hole where it can be put : put it over there

adv. :→ don't have to scan full list

disadv. → creating lot of small hole

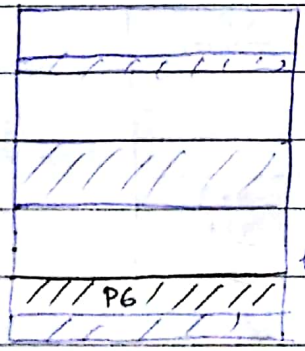


b) Best fit :

disadv. :- scan full list

adv. :- find best hole (minimizing no of holes)

c) Worst fit : giving largest hole possible



adv. : holes created are of larger size

a) In some cases, a) works better, b) works better in some c) usually never works better.

Teacher's Signature

→ Again compaction can be done ⇒ add. all changing again

⇓

Can do Runtime
Add. Binding Only

→ Compaction also has its own Overheads.

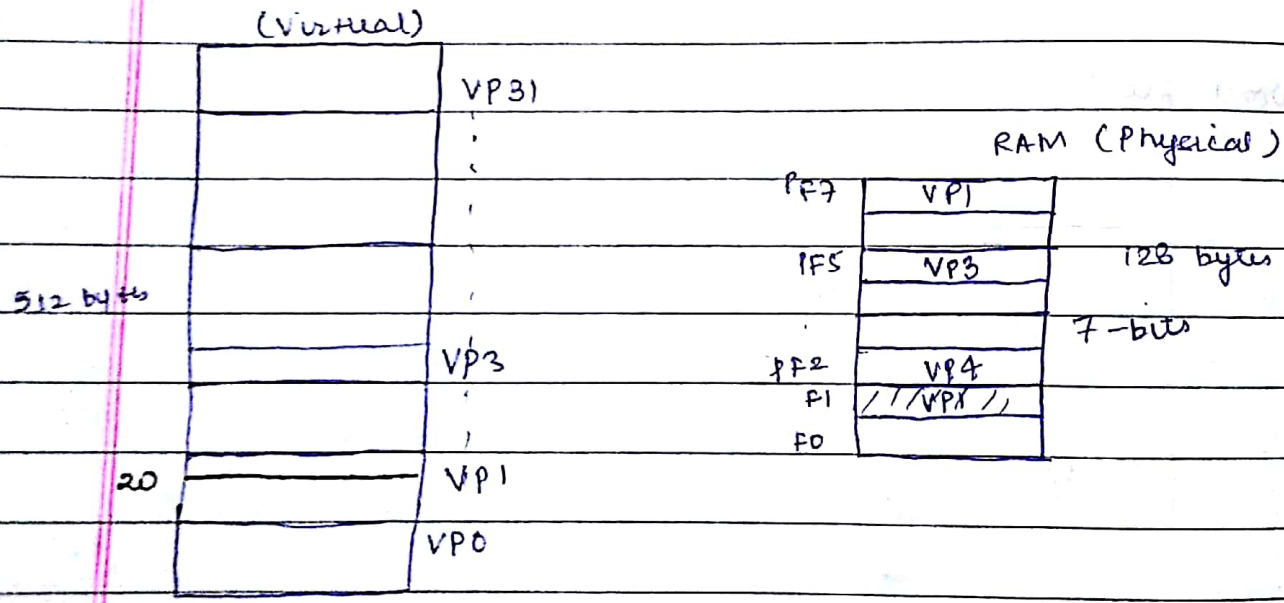
19/3/18

→ External fragmentation can take place in swapping area also (HD)
Doing compaction over there is not possible.

Good approach for memory management : Paging

Eg logical addrs (byte addressable) : 512 bytes ⇒ 9 bit add. is needed.

RAM : 128 bytes ⇒ 7 bits needed for phy. add.



→ 32 bit processor ⇒ CPU can have $2^{32}-1$ addresses (logical)

→ OS knows about phy. add. & ~~virt~~ logical add.

Teacher's Signature

Frame / Page frame

Suppose partition size : 16 bytes each

∴ No. of frames in RAM : $\frac{128}{16} = 8$

No. of virtual pages in V.M. = $\frac{512}{16} = 32$

(16 bytes each)

logical add : 20 → 0 0 0 0 1 0 1 0 0 — logical add. : 9 bits

I can either say logical add. 20

or I can say VP 1 offset 4 (20 - 16)

32 V.P. ⇒ 5 bits needed to specify V.P.

16 bytes each ⇒ 4 bits needed to give offset

9

0 0 0 0 1 0 1 0 0

Virtual Page No. | Offset

Add. 56 : VP : 3 Offset : 8

32

3 8 → 32 + 16 + 8

0 0 0 1 1 1 0 0 0

→ Many time, much part of code isn't needed. so, OS only loads that part of code in RAM which is needed.

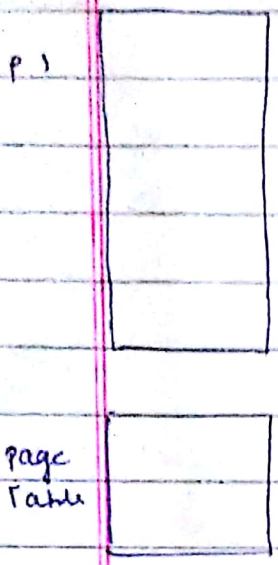
→ Suppose it loads VP3 in PF5

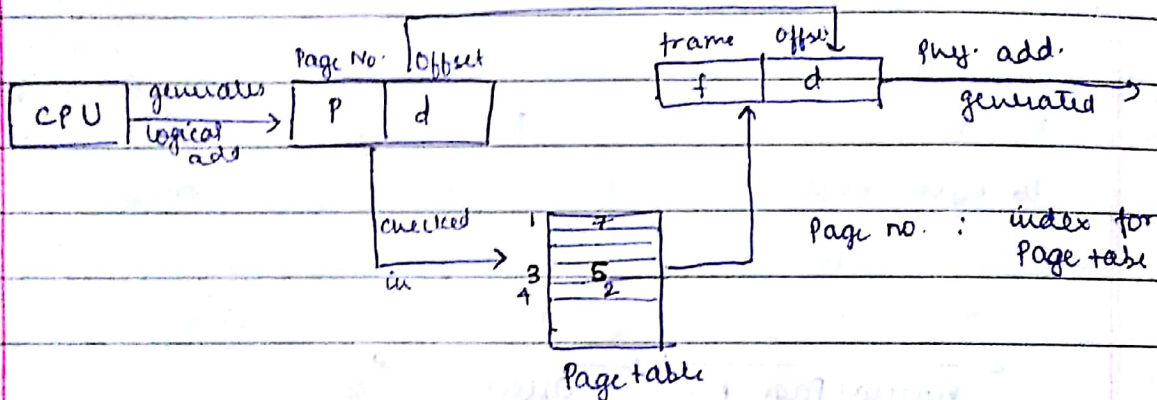
Now, PC is increment & PC has add. of VP4. It will pick VP4 & put it in PF2 (let say), When VP4 was running, there was a func^n call which was in VP1. So, now, VP1 will be picked & put in PF7. Return To VP4. Don't know add. of VP4. ∴ OS has to keep a

Teacher's Signature

Scanned by CamScanner

1) frame table which has list of all available frames in RAM.
2) for each process, it needs to maintain a page table.





VP 3 → check index 3 of Page table
↳ in PF 3 → 5 is stored at 3rd index of Page table

Phy. add : 1 0 1 | 1 0 0 0 → Offset will remain the same
(5 → Page Frame; 1 0 0 0 → offset)

Suppose add : 56 → VP 3 Offset : 8
VP 3 maps to frame 5

64 + 16 + 8
64 + 24 = 88 → phy. add. generated
→ 16 × 5 + 8 = 88

Teacher's Signature

typical frame size : 4KB

→ A: There could be multiple processes in RAM.

→ B: RAM is full ⇒ swapping will take place

→ At the time of context switching ⇒ state of process is stored in PCB

Similarly, here, PCB can store state here

How to decide Page size?

→ If too less : No of page ↑
No of virtual page ↑
⇒ Page table ↑ ⇒ no of mem needed

→ If too large :
No of Page ↓
No of V.P. ↓
Swapping will be ↑ (replacement ↑ hoga)

↳ If proc size : 500 bytes
* Last page me internal fragmentation of 12 bytes
On an avg. there is a chance of 0.5 internal fragmentation lost due to on last page (jo wo use kar rha h, uska last page)

⇒ here, page size too large ⇒ avg. internal fragment" ↑↑.

→ Assuming P1 & P2 in RAM :
When P1 is going to run, all registers will store its state
when context switching : store & save registers
This is possible only when page table ↓.

11 :→ RAM me har process ki page table store hogi.

→ Suppose P1 ki page table ka add. 100
P2 add. 150

Teacher's Signature

In CPU (or MMU), I will have a PTBR

Page Table Base Register

↓ stores physical add.

If P1 is running ⇒ PTBR → 100 ⇒ mapping will occur with Page table 1

P2 ⇒ PTBR → 150 (points to add. 150)

13/18

size(Phy mem) > size(VM)

→ Page size = 1KB

No. of bits req. for Offset = 10 bits = 8 d

L Add : has 4 pages ⇒ 12 bits req.

Phy Mem : " 8 " ⇒ 13 bits

→ No of entries in Page Table = 4 (4 pages in VM)

→ Size of each ~~table~~ entry in ~~table~~ page table

↓

Frame no.

⇓

3 bits (just getting frame no.)

It can access : $2^3 \times 2^{10}$ addresses (↑ offset)

→ If Page Table is too small : can use register for each page. On context switch, values of registers are modified

→ If Page Table is large ⇒ keep it in RAM only

Ki page table

Eg: P1 ~~page table~~ 100 - 180

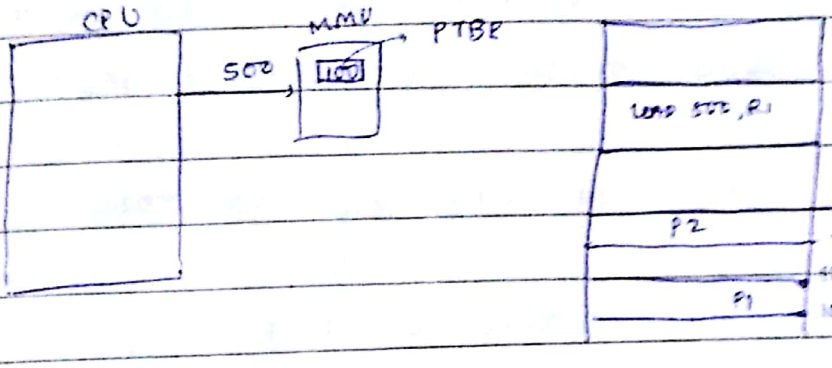
P2 " " " 200 - 250

↳ in P1

→ LOAD 500, R1 : direct add mode

↳ whatever is in add. 500, put it into R1 → user access

→ 500 : L Add. wala



500 is put into add. bus

MMU has PTBR. Since P1 is running, pointer for P1's page table is in PTBR

500 : | Page | d |

Check page table in which frame this page is in.

↓

go to that frame, add d, access it & bring data to CPU.

→ If 500 wala page isn't brought into RAM, it'll be brought from backing store & page table will be updated

→ If a memory access takes 100 ns

→ first access memory to access page table (100 ns) & get frame & then compute f + d, access that add in RAM (100 ns)

→ 2 memory accesses required for 1 inst^n.

Teacher's Signature

→ We don't want that

(doing that because Page Table is in memory)

To avoid this : Bring Page Table to Cache

However, won't use Data & Inst" cache here.

We have : Translation Lookaside Buffer (TLB)

↓

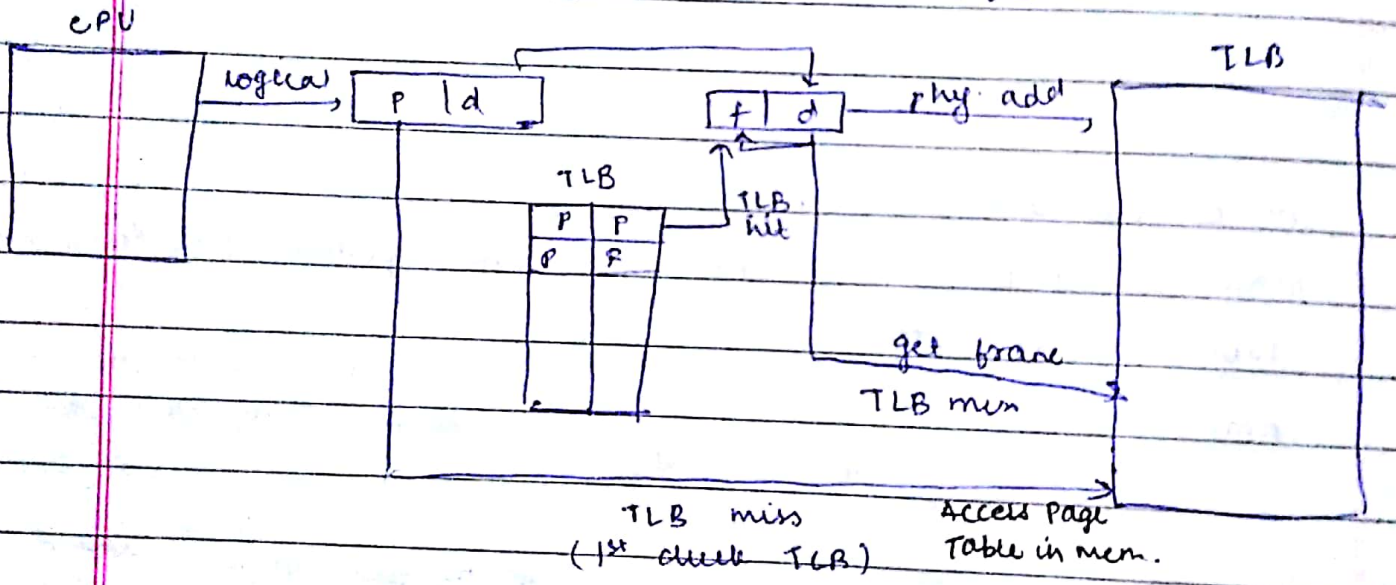
actually a cache

(could be in MMU or separate memory)

related to transl" of add. from logical to physical

It's small → can't ~~use~~ bring full Page Table to TLB.

1st time add. comes → check in TLB



If P_1 was running, 1st time there'll a miss. Once it's accessed, I'll bring some entries in TLB.

↓

has 2 values (atleast)

[Page No. & Frame No.]

* In original Page Table : It was complete : just needed frame (indexed into Page No.) no.

↓

There is no page no. here. Only has frame no. → If put into TLB → have to store page no. also.

(found entry in TLB

TLB hit

→ Suppose, we check TLB & get frame no. → add d & access (f+d)
that add

→ If TLB miss : have to access Page Table in memory, take frame no. & add d
↳ put in TLB also (either only this entry or a group of entries) (parallelly)

→ If hit rate = 80% → 80% of time, entry is found in TLB

Mem access : 100 ns

TLB " : 20 ns.

hit : 100 + 20 = 120

miss : 120 + 100 = 220
(First access in TLB → Loss)

effective mem. access time

= 0.8 (120) + 0.2 (220)

= 96 + 44 = 140

11B

→ Can have cache (L1) on the processor chip for faster access & L2 cache b/w processor & RAM.

① (TLB)
→ In cache, apart from tag, we've valid / invalid bit & also present

	tag	valid/invalid bit	Read / Write permission
0	10	0	
1	1	0	
2			
3	4	1	
4	15	1	
5	10	1	
6			
7	12	0	

Initially, when not brought into RAM, all bits = 0

Page table in RAM

Suppose Page 4 was brought & put in frame 15

⇒ valid bit = 1

At execution time:

Check Page no. 4 ⇒ also check valid bit

if = 0 ⇒ invalid ⇒ The page is not brought in RAM

or I'm trying to access an add. which isn't map in my add. space

If → in my logical add. space ⇒ bring page to RAM & modify bit = 1

If → Not in my " ⇒ Trap to OS (error cond" will be generated)

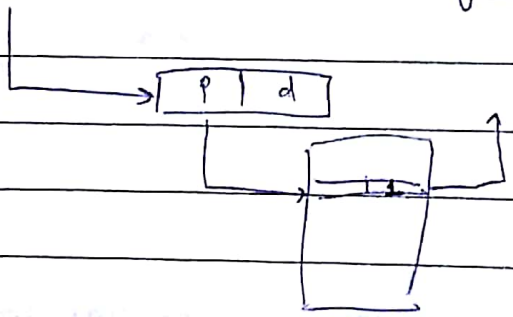
② Next is Read / Write / Execute permission

also stored in Page table

log. add.

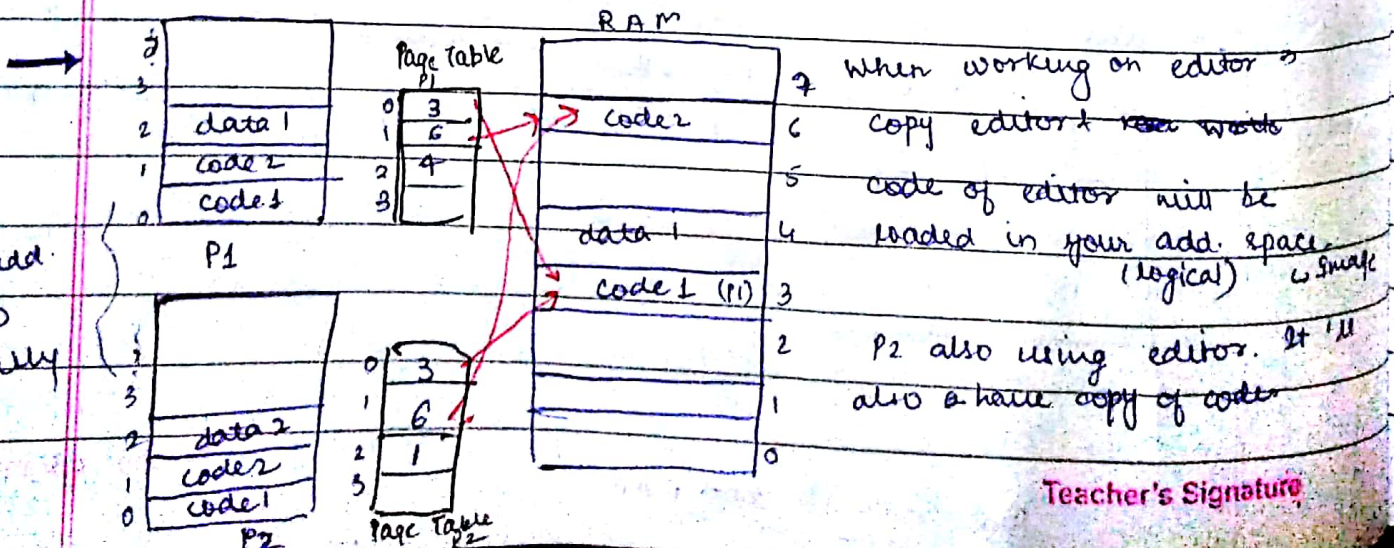
STORE R1, 2500 = store value of R1 in 2500 mem-add.

WRITE Opⁿ



→ Write bit = 0 ⇒ Only read & execute permissions.

If want to write (STORE) → again an error.



When working on editor ⇒ copy editor & code of editor will be loaded in your add. space (logical) ⇒ Image

P2 also using editor. It'll also have copy of code.

Teacher's Signature

Editor → type of app^s are shared

DATE: / /
PAGE NO.:

Now, P2 also needs to be loaded. But OS sees code1 & code2 are already in RAM.

code1 & code2 → shared

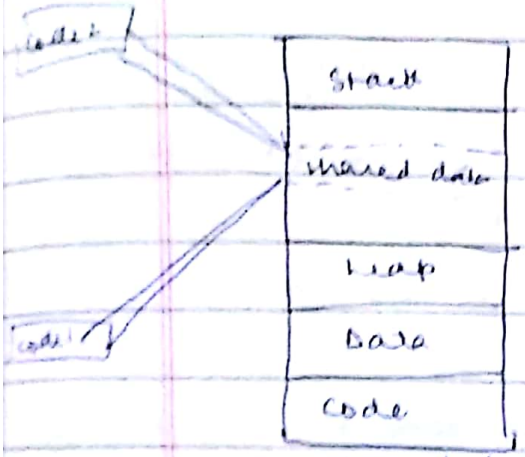
→ Map code 1 → 3 & code 2 → 6 Only data part'll be diff

3 & 6 : Shared Pages

Shared memory — Pure Pages . can't be changed

⇓

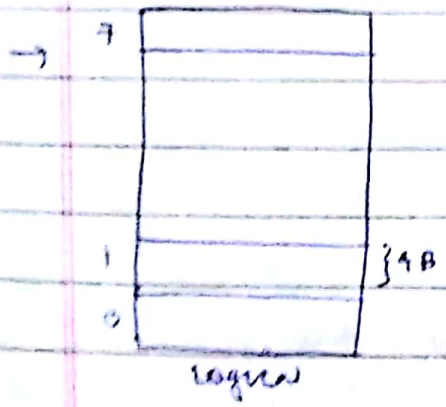
Only Read mode (no write mode)

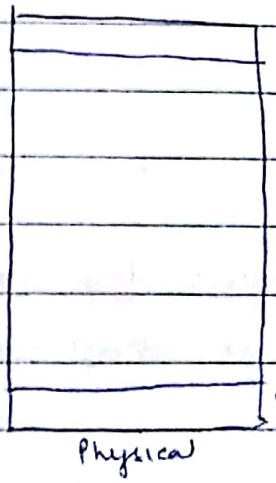


* Separate for P1 & P2

Phy. Mem. will have just 1 copy of it.

Log. add. will be diff for both





Logical add : 8×4

= 32

⇓

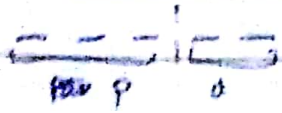
5 bits needed for add

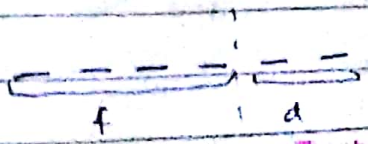
⇓

64 Bytes

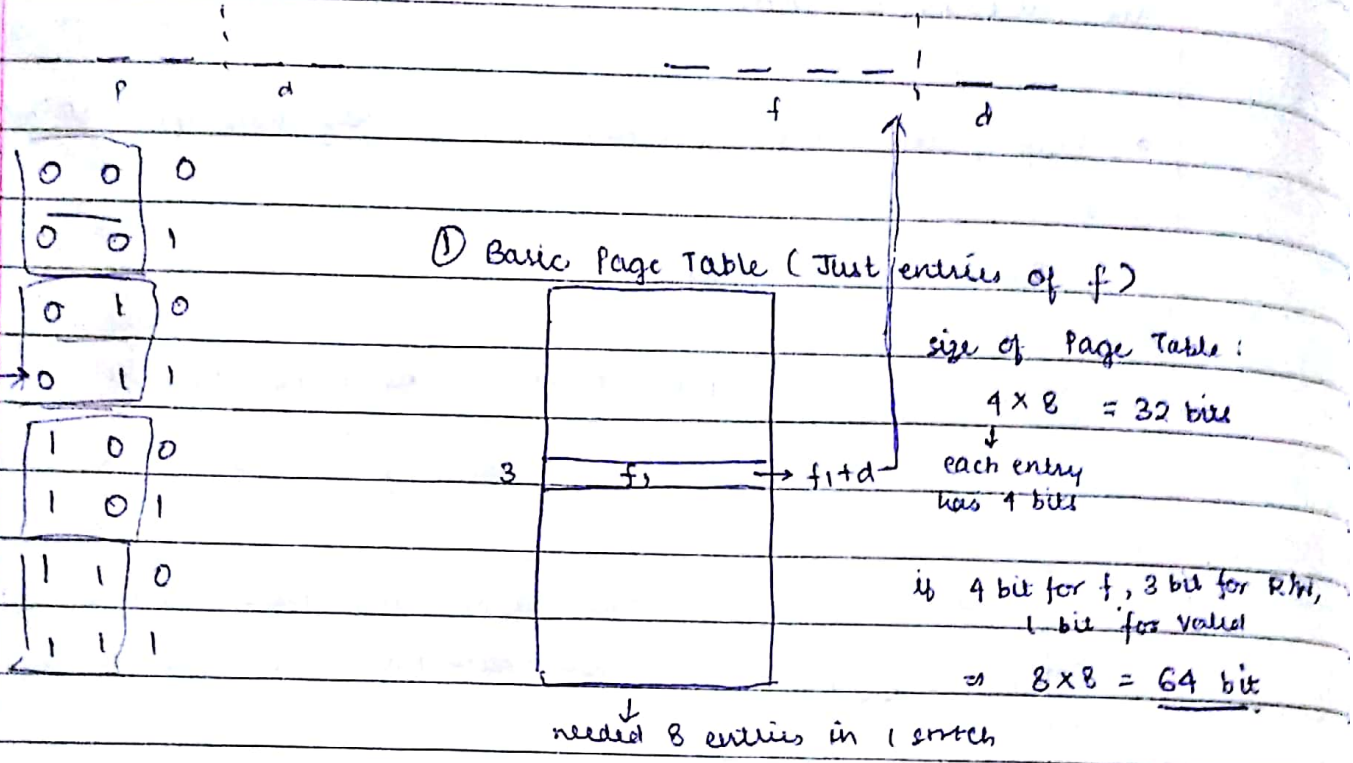
⇓

6 bits ∴ for add.





Different structures of Page Table



① Basic Page Table (Just entries of f)

size of Page Table :

4 × 8 = 32 bits

↓ each entry has 4 bits

if 4 bit for f, 3 bit for RWI, 1 bit for valid

⇒ 8 × 8 = 64 bit

Basic Page Table : Linear table (starting se end tak)

→ Suppose 9 have 1 million pages → Page Table has 1 million pages

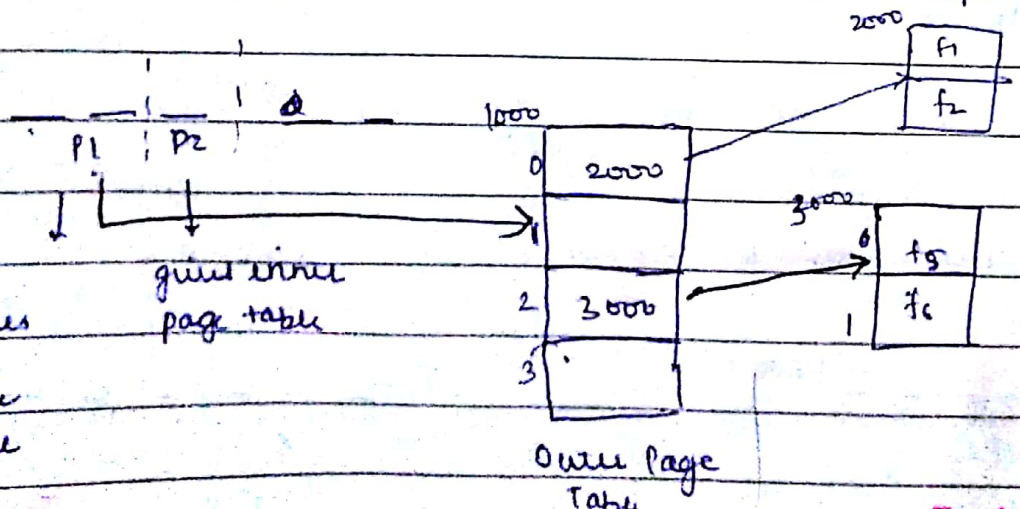
→ Contiguous space needed in RAM : very large space

↓

Break Page Table in RAM

② Hierarchical Page Table

Break page part (here : 3 bits) into two parts



Teacher's Signature

Now, PTBR will store base add. of Outer Page Table (1000)

If CPU : 0 of P1 Outer will have entry 2000

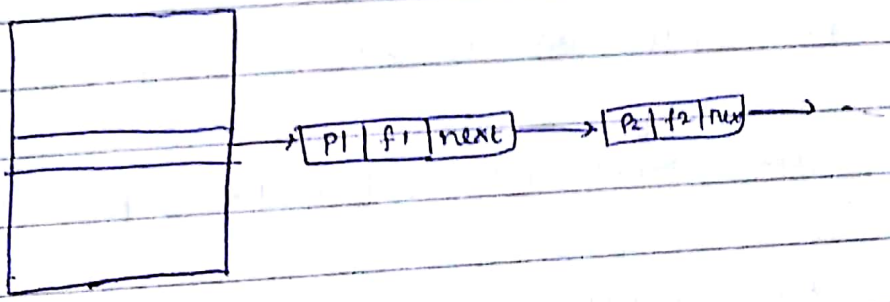
- have pvt 4 entries, 2 entries, 2, 2, 2 somewhere else & have ptr to ptrs

→ If 101 : 2 of Outer → fr f4 → f4 + d

Add. is still larger → can make 1 more layer

③ Hashed Page Table

→ can have Page table in form of linked list in case of Hash func^n (using page no.)

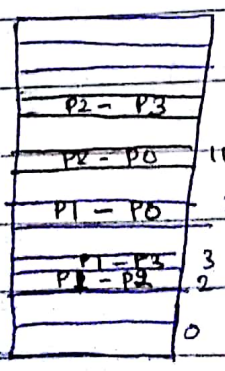


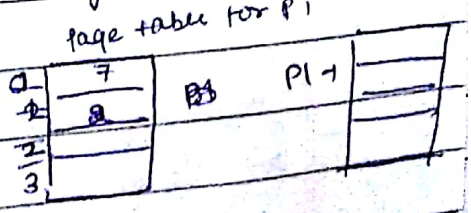
④ Inverted Page Table

Right now, we've 1 Page Table per process.

Now, we'll have 1 Page Table for complete memory.

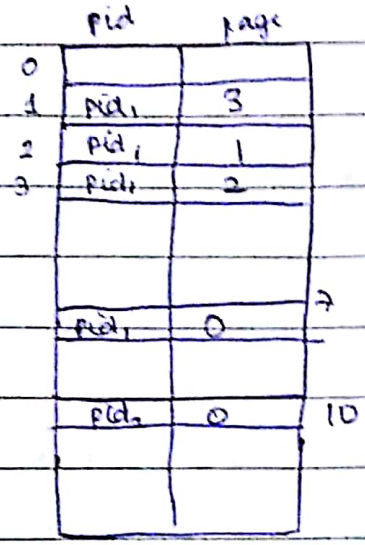
Mapping will be from Frame → Page





Teacher's Signature

→ Earlier, had different page table for different processes.
→ Rather than keeping track of pages, we will keep track of frames
→ Other way : have 1 common table
↓
will have all frames in RAM
Mapping : Frames → Page



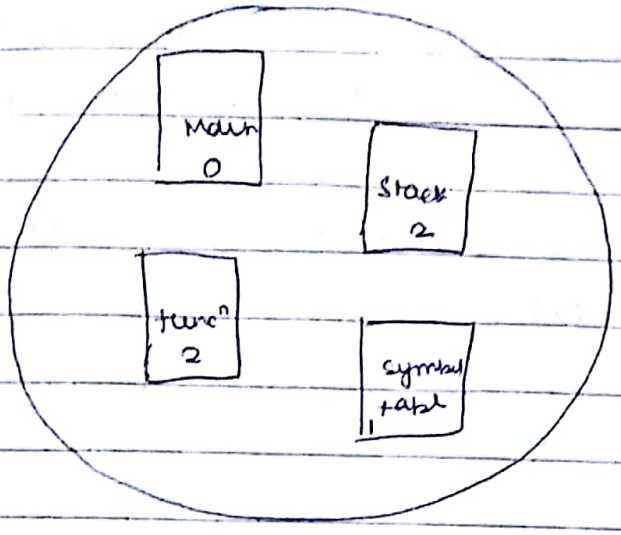
→ Inverted Page Table

Now, when logical add. is generated : pid, page & offset will be generated
↓
check whole table, find pid & page → index will give me frame
frame + Offset
↓
give as physical add.

✗ This can't be used in case of Shared Pages. (1 index can't have 2 entries)
→ In this case, if you've shared code, you'll have to keep multiple copies of the code.

→ Searching will become complex if no. of frames is too large

User view (Synonym to User view of a particular app^n)



→ segmentation

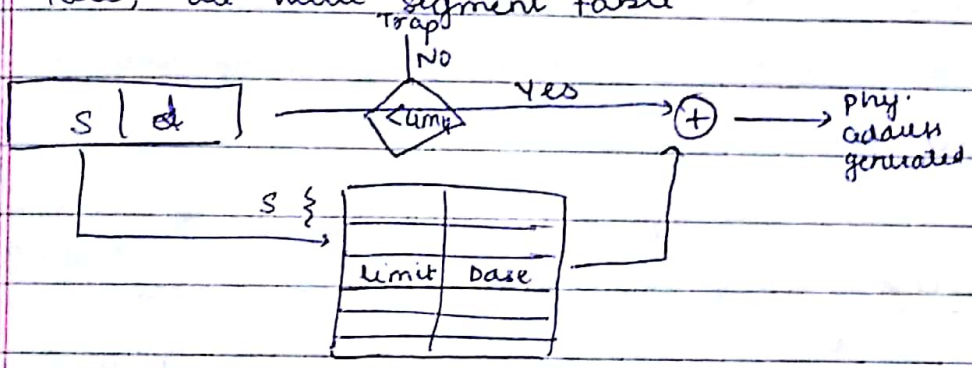
logical add. generated is of form

| s | d |

→ it's not linear form of addresses (like pages) → fixed size

→ very similar to paging : all segments can be of diff. size.

Here, we have segment table



Segment Table

Each segment is going to be loaded somewhere in memory

→ Have to keep track of base & limit of that segment.

if d > limit
↓
trap to OS

↑ d will be checked by limit

phy. add = base add. + d

→ If I used paging ⇒ all segments will be in pages.

(½ main in some other frame, ½ main somewhere else)

Here, all inst^{n}s in main are available together

Teacher's Signature

→ Not have to jump again & again from 1 frame to another.

Disadvantage: → If segments are too large : may have to swap with some other process (Replacement Algo)

→ Many systems use comb" of Segmentation and Paging.
(Divide segments into pages)

Advantage: → more synonym of user view

→ Each approach will have its own advantages and disadvantages.

⇒ Suppose 25 Pages are there in ~~the~~ VM. → P1

Many times, some routines are to be called only when there is some exception.

Should be brought in RAM only when required ⇒ Demand Paging

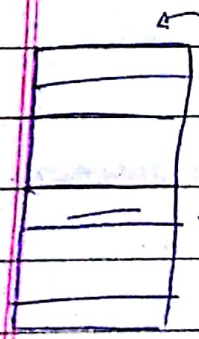
Demand Paging → all ~~the~~ pages available in swap area

P1 is about to execute. → in Backing store.
(in swap area → in Ready Queue)

↓

processes selected to be in readyQ

Jobpool : who want to go to RAM & get CPU



156 pe → 1st instn : checked Page table : Valid bit = 0 → page not in RAM

↓

No frame entry available

& → It is needed now.

⇓

Page fault has occured : when required mem reference is not available in RAM.

Now, what will happen ?

→ Disk has to be accessed

→ Page has to be brought into RAM.

Page Fault : give msg to OS.

→ OS will give request for Disk access. (device access : Process can't use it directly)

⇓

this Process goes in device Queue for HD ⇒ waiting for some I/O.

→ while this process is not running, CPU will be assigned to some other process

→ Meanwhile, page will be located in disk & trans

→ look for free frame in RAM & bring page.

→ Update Page Table for that process.

→ If no free frame ⇒ some frame has been taken out

⇓ update its page table also.

→ ~~to~~ Table maintained by OS (of all processes) also needs to be updated info.

→ Process 1 is again in Ready Q with updated PT

→ TLB ki PT also needs to be updated.

→ ~~May be~~ CPU scheduling also occuring

→ May be some replacement algorithm is also working.

→ Agar ~~sirf~~ sirf 1 hi process h : No need of demand paging

↳ directly pages ko RAM me load karde.

Only In Demand Paging :

* All Pages are in backing store : bring required page to RAM.

→ No. of Page Fault v/s degree of multiprogramming :

This is calculated

Teacher's Signature

accessing RAM : 100 ns

cache : 10-15 ns

HD : ms

→ If P1 → 25 pages & RAM size = 50 pages

Degree of multiprogramming : 2

→ Suppose Page fault rate = p

Memory access time = ma

How many times page will be available in RAM ? 1-p

Eff. Mem. access Time = $(1-p)ma + p \times$ page fault time

(1-p) ma → ns

page fault time → accessing the HD (ms)

If page fault rate ↑ ⇒ Eff. Mem. access Time ↑↑.

⇒ have to minimize p.

1) Can ↑ no. of frames : Page size ↓↓

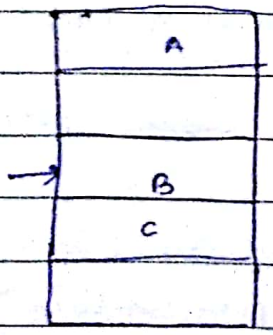
⇒ size of page table ↑

⇒ Internal fragmentation

↑ can cause lot of page faults.

↳ If a frame alloc" algo.

↳ If all frames are full → need frame replacement policy



B → goes back into Backing store

⇒ whole page is to be written into Backing store

↳ will take some time.

(Disc access time + writing back in HD)

Teacher's Signature

To minimise this access time

→ can use Dirty bit / Modified bit.

With every page : have " . any write opn

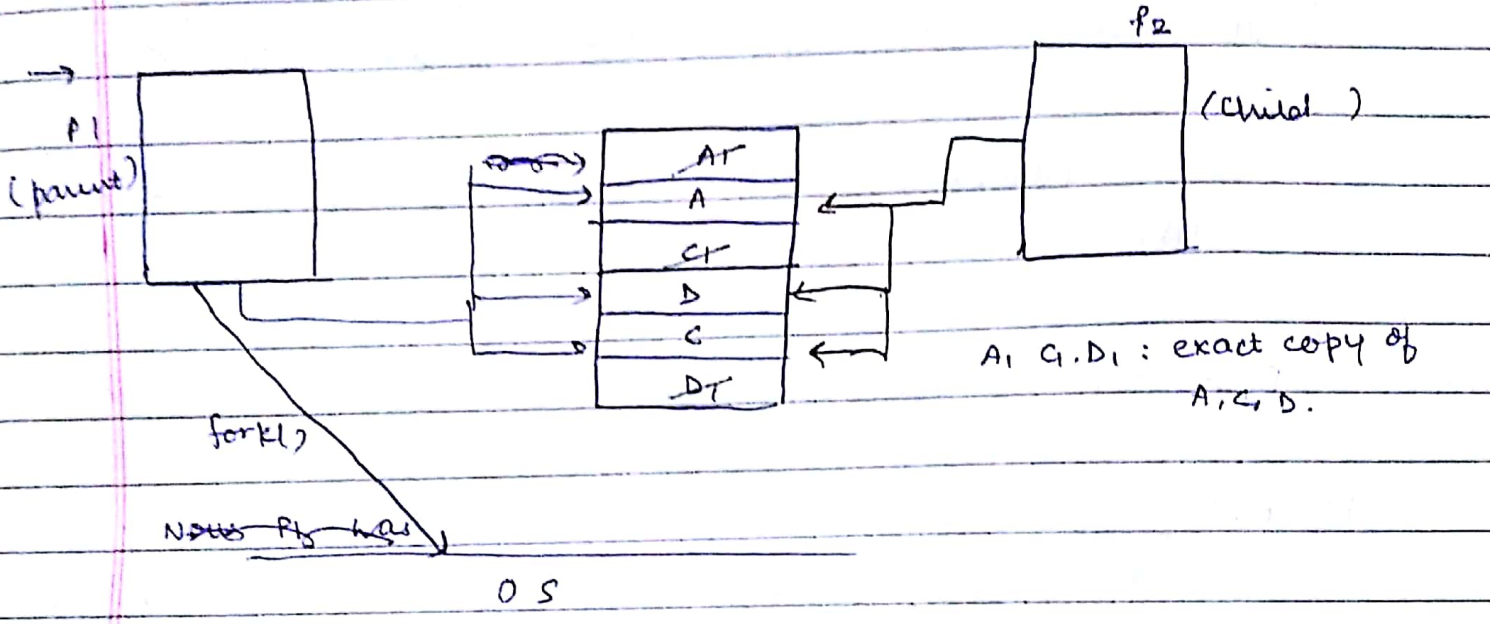
If I've made any change in page in RAM ⇒ dirty bit = 1.

When replacing, check bit into

If bit = 1 ⇒ need to write backing store

bit = 0 ⇒ Original copy is already in HD.

Simply overwrite it by bringing new page.



Now P1 has given fork() system call.

↳ child process is created with exact replica.

→ When new code is written in P_2 : new pages 'll come in RAM at A_1, B_1, C_1.

→ If not ⇒ we're wasting frames putting 2 exactly same pages

⇓

It asks child also to share the same pages.

P_1 & P_2 both are running at same time.

- When P_2 wants to change a page, then only OS will make a copy of that page & P_2 will make change in that so page of parent process doesn't change.

This technique is called : Copy - on - Write (*Imp)

child will get its own copy of pages only when it needs to make a new copy & make changes in that only.

→ Even if parent is making change, copy will be created for child.

Page Replacement Algorithms :

1.) FIFO

7 0 1 2 0 3 0 4 2 3 0 3 2 1 → Virtual Page no.

Only 3 frames in RAM. Count page faults

7	F (Fault)
7, 0	F
7, 0, 1	F
2, 0, 1	F
2, 0, 1	X
2, 3, 1	F
2, 3, 0	F
4, 3, 0	F
4, 2, 0	F
4, 2, 3	F
0, 2, 3	F
0, 2, 3	X
0, 2, 3	X

7 0 1

0, 1, 3 F

3 frames → 11 F

1 frame → 14 F

Teacher's Signature

DATE: / /
PAGE NO.:

If ↑ frame no. ⇒ no. of F ↓

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

3 frames : F F F F F F F X X F F X ⇒ 9 F

4 frames : F F F F X X F F F F F F ⇒ 10 F

↳ Even after ↑ frame no. ⇒ no. of F ↑ : Belody's Anomaly

2.) Optimal Page Replacement Algorithm :

Replace that page which will not be used for longest period of time ⇒ looking into future.

→ same eg :

7	F
7, 0	F
7, 0, 1	F
2, 0, 1	F
2, 0, 1	X
2, 0, 3	F
2, 0, 3	X
2, 4, 3	F
2, 4, 3	X
2, 4, 3	X
2, 0, 3	F
	X
	X
	F

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

→ 7 isn't used for longest period of time

→ Replace 1

3 frames : 8 F

⇒ can't look into future. ⇒ need to know complete reference string.

Teacher's Signature

→ this is just used for comparison with other algo (optimal page)

3.) LRU :

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

7	F
7, 0	F
7, 0, 1	F
2, 0, 1	F
2, 0, 1	✗
2, 0, 3	F
2, 0, 3	✗
4, 0, 3	F
4, 0, 2	F
4, 0, 3	F
0, 0, 3	F
0, 3, 2	✗
0, 3, 2	✗
0, 3, 1	F

3 frames : 10 F

→ most commonly used

→ Implement : 1.) Put time of use in PT for each page when page is used (time of use) → replace that page with min time of use

(use counter)

2) use stack : Bring a page : push (doubly linked list)

TOS : most recently used

use doubly linked list. already had 0, again for 0 used : remove 0 & push on top

Page at BOS : least recently used

→ causes a lots of overheads.

2/14/18

4) **LRU Approximation :**

→ can have LRU approximation : keep reference bit (following LRU)

if 1 ⇒ used recently

0 ⇒ not " "

→ may use Additional bits (Ref. bits)

In page table : have ref. bits

Let → 3 bits — 0 0 0 → Initially

when pages in brought

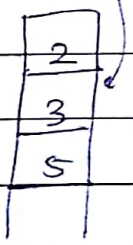
page 3 — 1 0 0 → 010 → 001 (shift)

page 5 — 1 0 0 → 010 (shift)

1 0 0 (shift)

whichever value is least ⇒ it is LRU ⇒ has to be replaced (reference bit)

2 3 (2) 5 (3) 0



because 2 has come

0 0 → 0 1 0 —2→ 0 0 1 —5→ 0 1 0 —3→ 0 0 1

1 0 0 —2→ 0 1 0 —5→ 0 0 1 —3→ 0 0 0

1 0 0 → 0 1 0

if referenced, put 1 here.

Page replacement : check value of reference bit)

↓ → which page will be replaced ?

→ 23524 when 4 comes

2	100 → 3, 010 → 5, 001 → 000 → 4
3	100 → 5, 010 → 2, 001 → 1
5	100 → 010 → 2

→ Page 3 will be replaced when 4 comes.

5) Second chance Algorithm :

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

F F F ✗ F ✗ F ✗ ✗ ✗ F ✗ F ✗ F ✗ F ✗ F ✗ ⇒ 9 F : 18 references

↳ If I'm going to attach a ref. bit (2nd chance bit) to each page.

↳ Everytime new page is brought bit = 0

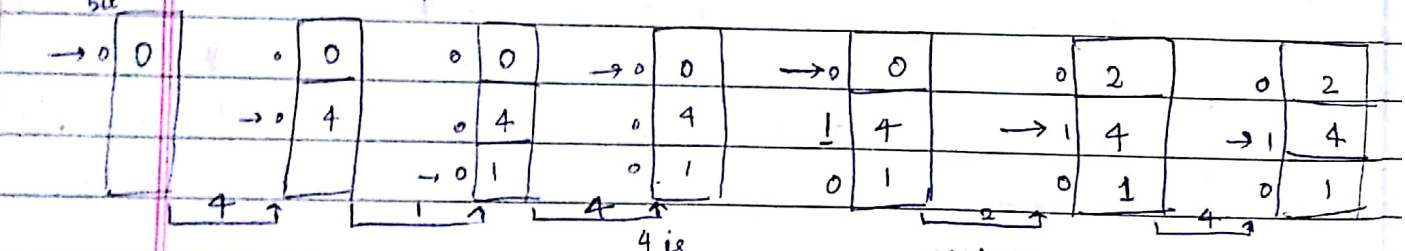
↳ " page is referenced, ref. bit = 1

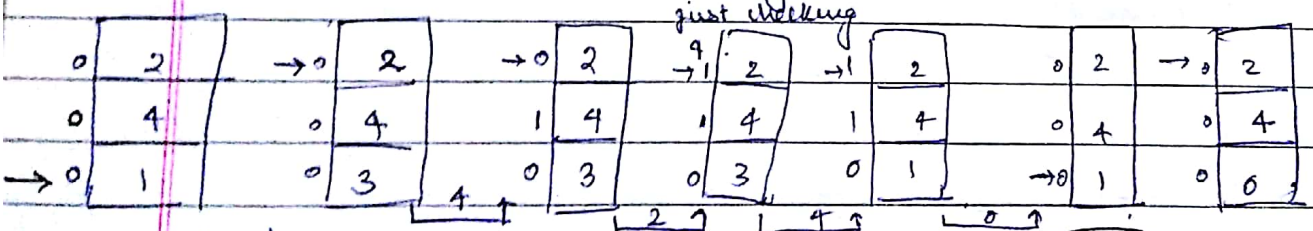
↳ If a page is to be replaced, we will go in Round Robin manner & check ref. bit (starting from ptr)

If bit = 1 → make bit = 0 : got a second chance.

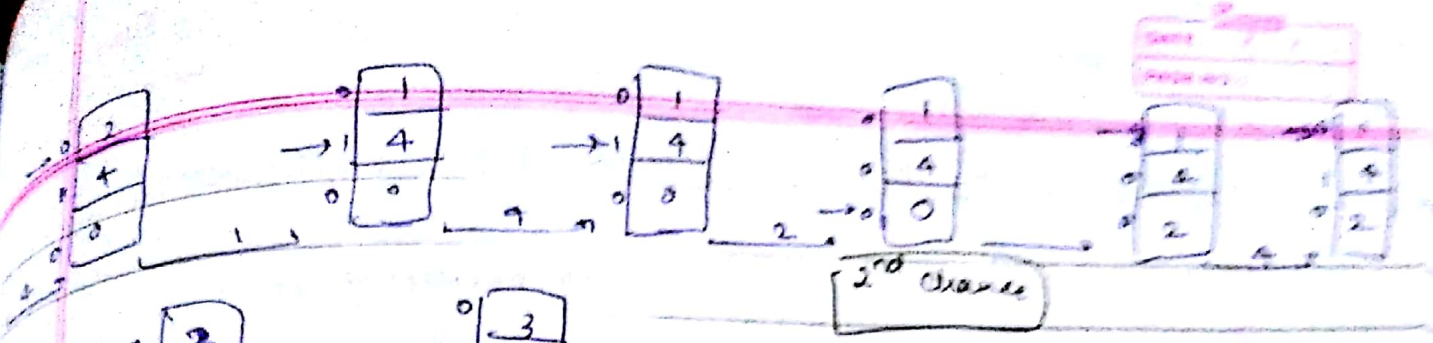
= 0 → replacem that page

1 means it's recently used bit





Teacher's Signature



→ If it had been a regular algorithm, 2 would've replaced so many times.

Backing store : In Disk (Sometimes, called Paging Disk)

→ When a page is replaced (1 by 3) → 1 has to be written back into disk assuming dirty bit = 1

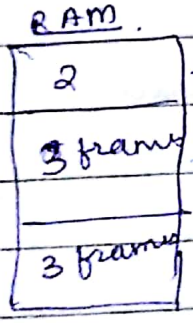
OS keeps pool of free frames

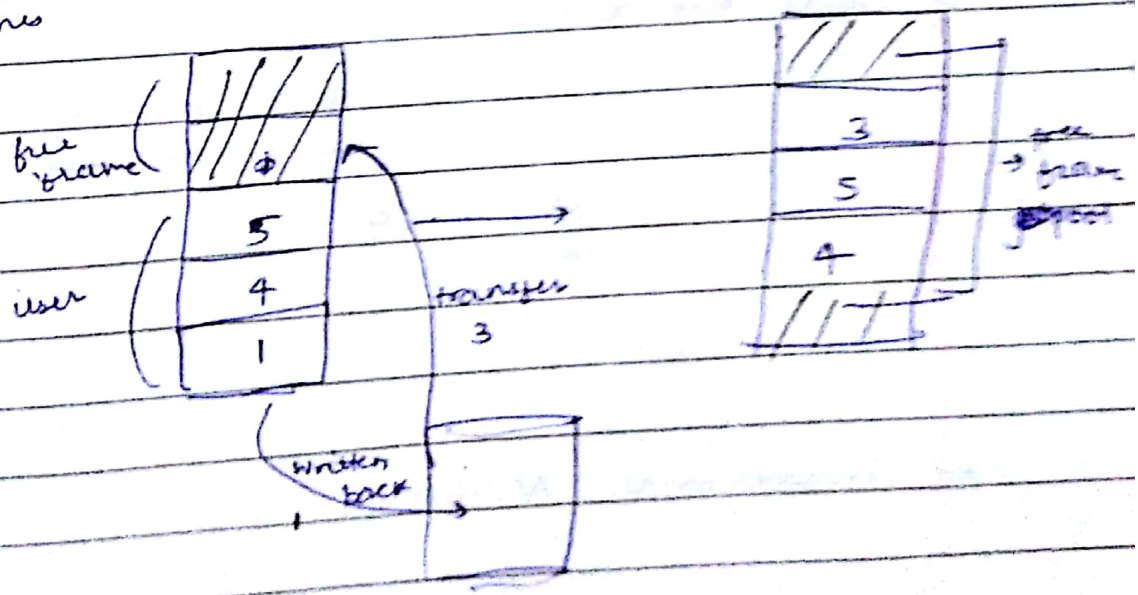
↳ First write 1 in disk, then only transfer 3

Now, while 2 is being written in paging disk, transfer 3 in free frame pool

↓

doing parallely



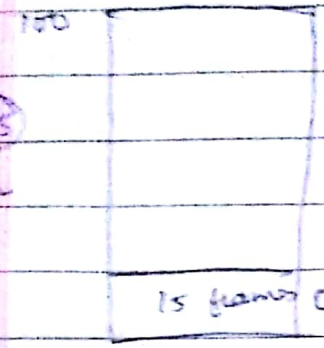


Free frame table / is also to be maintained regularly

Page table

Teacher's Signature

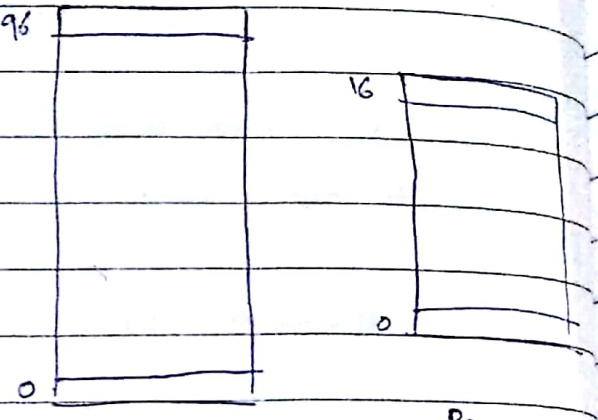
Allocation of frames (Assume no free frame pool)



No multiprogramming

⇓

1 process in RAM at any time



P_1 is running & demand page is executing

logical space for P_1

→ Total 85 references : no problem
to diff. pages

→ After that : Page Replacement will occur.

↳ If Multiprogramming allowed ?
How to allocate frames to processes ?

1) Equal Allocation :

$$n = \frac{m}{p} \quad \begin{matrix} \longrightarrow \text{no. of frames} \\ \longrightarrow \text{no. of processes} \end{matrix}$$

$$= \frac{85}{2} \simeq 42$$

↳ Not sensible

2) Proportional Allocation :

$$S = \sum s_i = 97 + 17 = 114$$

Teacher's Signature

$$m_1 = \frac{a_1}{S} \times m$$

$$m_1 = \frac{90}{114} \times 85 \quad ; \quad m_2 = \frac{17}{114} \times 85$$ (we approximate value to int

(Total can't exceed 85)

→ No. pages allocated in RAM > pages required by that process (min no. of)

how do we know this