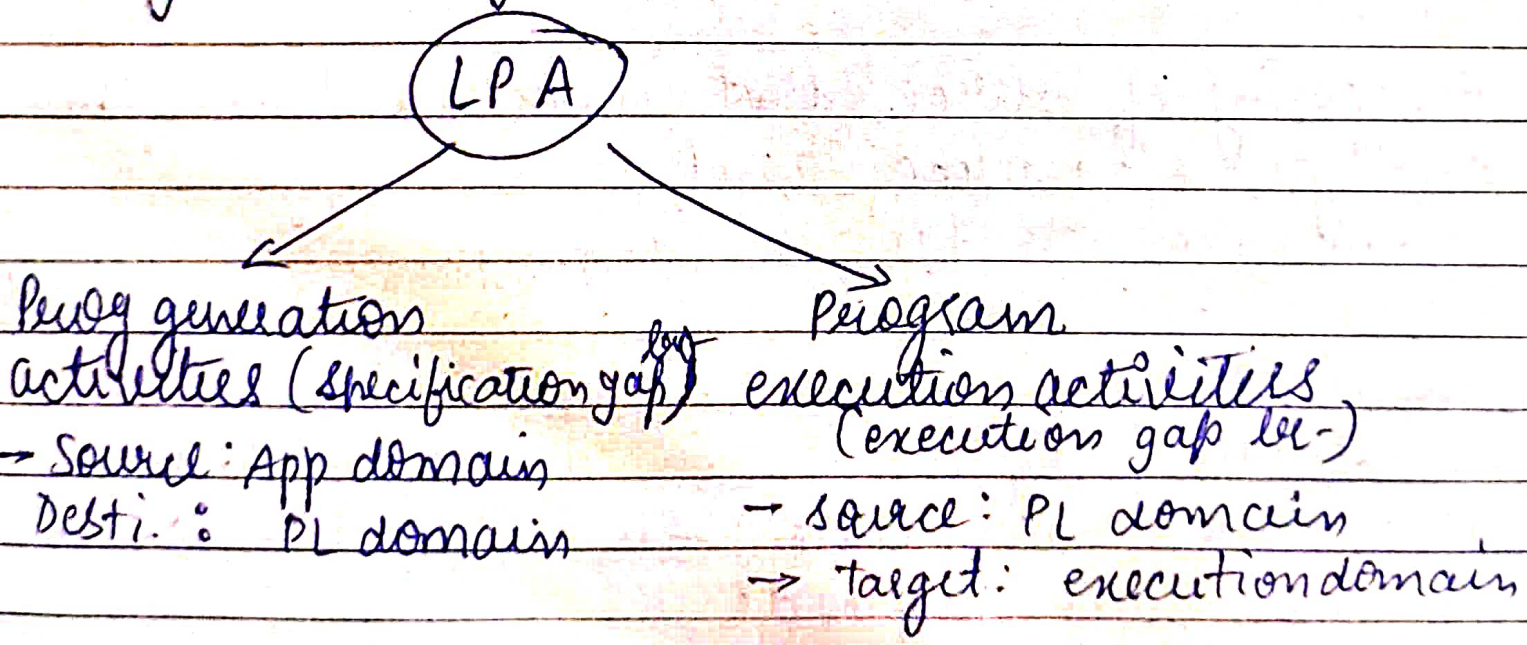
Language Processor

→ not possible for us to write in machine lang for comp. better understanding
→ Thus we use h-L-L like C++, Java. But their source codes can't be executed directly by computer
→ Lang Proc. converts (+translates) ~~and~~ source codes into executable machine codes

Lang. Processing Activities



Date: / / Page no.

1. Prog. generation: generates prog. acc. to specs. in target PL.
→ prog. generator is s/w that accepts instructions from individual to generate prog

2. Prog Execution → executes program for user in execution domain

models

Translation
I/p: source prog.
O/p: target prog.
→ translates source program written into machine lang.
H.LL → M.Lang.

Interpretation
→ reads source prog and stores it
→ takes statement, det. its meaning and performs actions
→ CPU uses PC (prog. counter) to note address of next instruction
P: Fetch → decode → execute

LP.D.T
→ Processing of source text in source languages
→ analysis of source text
→ translation to target languages.

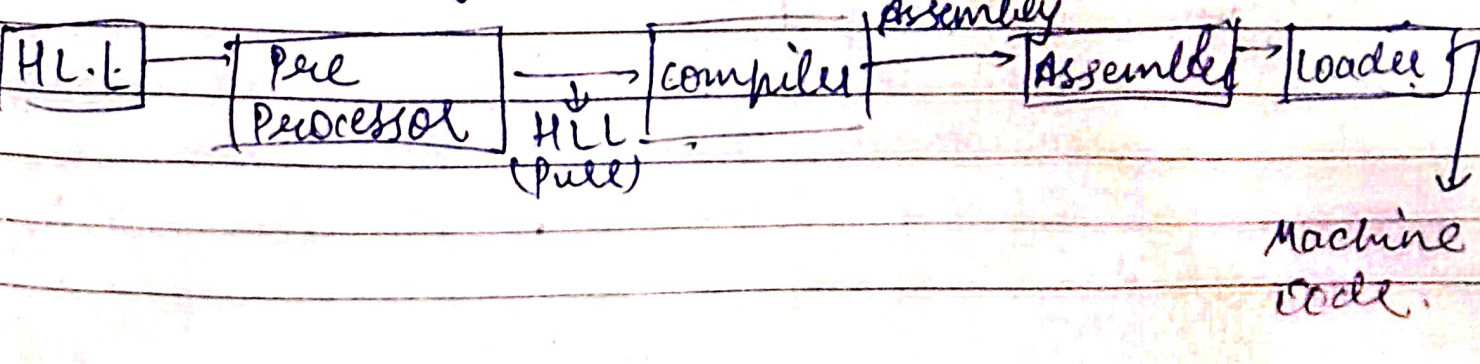
Assemblers
→ translates prog written in assembly lang into machine lang. code.
→ I/p is assembly lang code
→ O/p is machine code understandable by comp.

Compilers
→ processor that converts HLL code, into machine codes.
→ I/p is High level lang taken as a whole
→ O/p is Machine code eg. to HLLang prog
→ task is performed only if source code is free of error.
(C, C++, Java)
(time↑↑, generates Intermediate obj)

Interpreters
→ process that converts HLL code into machine code line by line
→ founds any error then terminates execution
→ moves only when error is removed
(Python, Matlab)
(time↓↓, no Intermediate obj)

Macro Processor
→ Instructions that are notational convinience for programmer
→ For every occurence, whole macro body gets expanded into main source code
→ Macro processor replaces each macro instruction with its corresponding source code
→ Involves Definition, Invocation and Expansion
(DIE)

Compiler Design



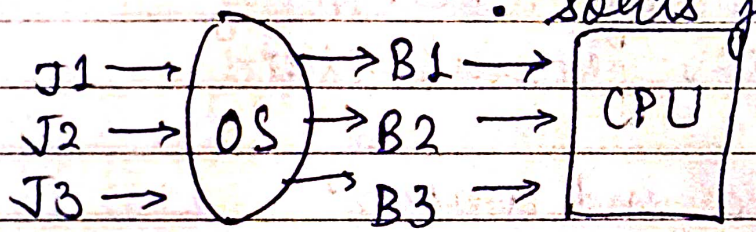
⑤ Distributed system
→ communication through networks is made easier

⑥ Special purpose system

⑦ Provides nice computing env.

Types of OS

1) Batch Operating System
→ no direct connection with comp.
→ Role of operator : groups similar jobs in batches
: sorts jobs with similar needs



Adv
- Ease in knowing time completion of job while in queue
- Multiusage of batch
- less idle time
- easy mgmt.

DAdv
- hard debugging
- costly
- failure of one job would make other jobs to wait
- operators need to be familiar with BS.

eg: Bank statement, Payroll system.

Date: / / Page no:

2) Time Sharing OS (Multitasking OS)
→ every task is given particular time for execution
→ tasks can be from single / multiuser.
quantum: time each task gets

Adv.
→ every task equal opp.
→ ~~redundancy~~ reduction in s/w duplication
→ less idle time

DisAdv
→ not reliable
→ D com problem
→ data not secure

Real time OS
→ serves real time system
→ processing need very small time interval
→ response time: small TI.
→ used for strict time req.

Hard RTOS
→ very strict for time constraint
→ needed to avoid accidents

Soft RTOS
→ less strict for time constraints

Adv
→ max utilisation
→ less time in task shifting
→ more focus on app
→ error free
→ best memory allocation

Dadv.
→ limited task foundation
→ expensive
→ algo complexity

Eg: Scientific exp, medicals, robots etc

4) Distributed
→ modern OS
→ a user can access other resource files connected with n/w

Adv
→ one failure doesn't affect another.
→ Email inc. data xchange speed
→ loads on host comp reduces
→ reduced delay in data process
eg: Locus

Dadv
→ failure of main n/w stops entire comm.
→ not well defined langs are used.
→ expensive.

5) Multiprog. OS
→ chooses one job from pool of jobs keeping other in waiting.
→ small memory can be used.

6) Parallel OS (Multiprocessor)
→ ~~at~~ using two CPU with single computer system
→ all CPU shares Bus, memory and other PPS
→ tightly coupled
→ systems are used where large vol data needed at high speed

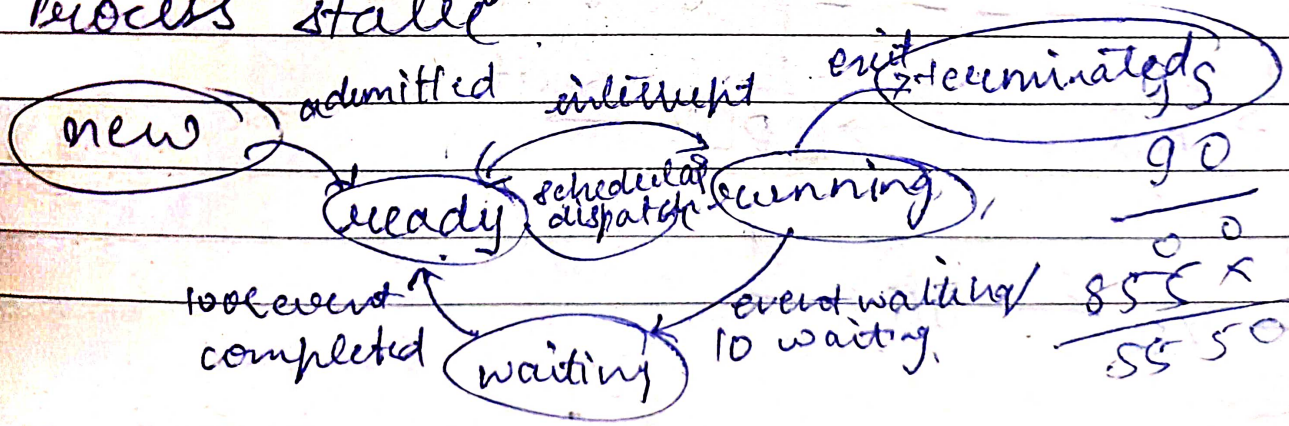
Process : eg.

↳ Passive entity depicting files that contains set of instructions set on a disc is called program.

↳ Process is active entity with program counter for next instruction to execute and set of resources.

→ process is program in execution
→ instance of running.

Process stated



Process Control Block (PCB)
→ unique to every process
→ maintained by OS
→ identified by Integer process id.
→ needed to track process

PState
P Privileges
P I d.
Pointers
Prog counter
CPU register
CPU scheduling
mem. mgmt Info
Acc. Info.
Status Info

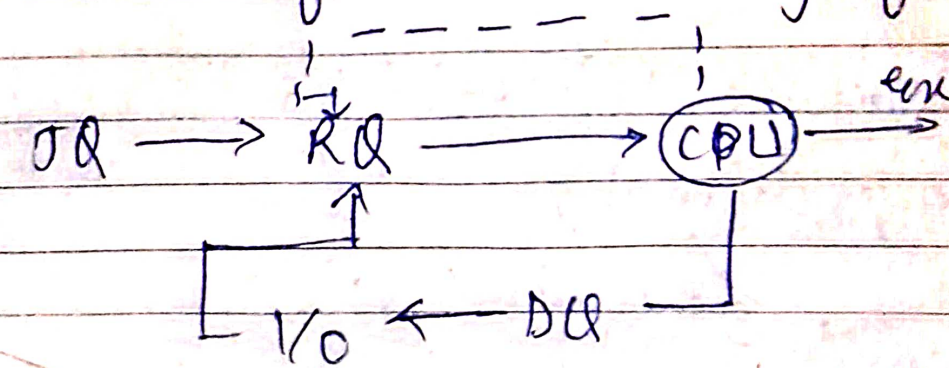
Process scheduling.

Process scheduler schedules and selects an available process for execution.

→ removal of running process from CPU and selecting another one on basis of some strategy
→ done for meeting the type of OS being chosen

Process scheduling queues
→ OS maintains PCBs in PSQs
→ 1 State = 1 queue
→ change in state = change in queue

① Job queue: keeps all processes in system
② Ready queue: keeps set of processes in M.M., ready and waiting for execution
③ device queue: processes blocked ~~by some~~ because of unavailability of I/O device



Date: / / Page no:

5 state process model

Running	Non Running
→ actually executing	→ waiting for execution
→ new process by default comes here	→ interrupted, waiting for I/O

Schedulars

→ handles scheduling in various ways
→ decides which process should run

Long (JOB schedulars)	Short (CPU schedular)	Medium (process swapping)
→ slow → controls degree of multiprog. → selects process into pools	→ fast → provides lesser control over degree of multiprog. → selects ready process	→ medium → reduces degree of multiprog. → reintroduce process into memory

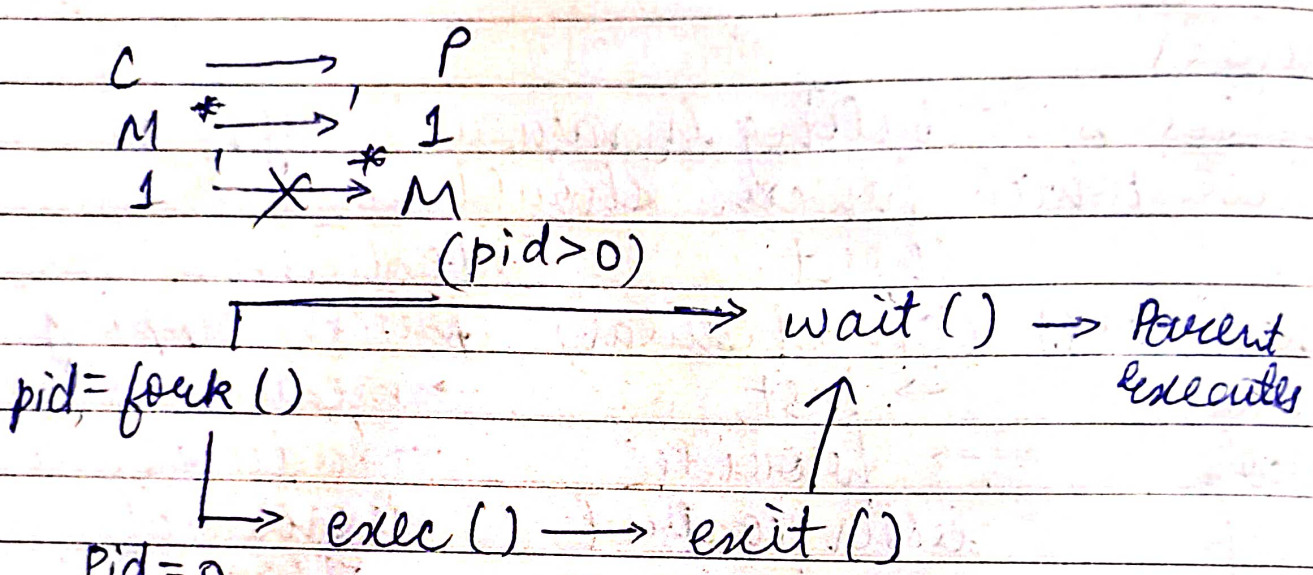
Context switching

→ when process is suspended due to interruption, it context must be saved so as to resume same process further.
→ state & context is saved in process control block.
→ after suspending and saving context of 1 process, CPU is switched to another process that is called context switch
→ highly dependent on h/w support.

Date: / / Page no:

Operation on Processes

① Process creation
→ created using fork() by another process
Parent process : creating
child process : created



② Process preemption
→ interrupt mech
→ running process is suspended
→ next process is determined and executed
→ makes sure all process get CPU time

③ Process Blocking
→ blocked as it is waiting for event
→ after event it goes for execution

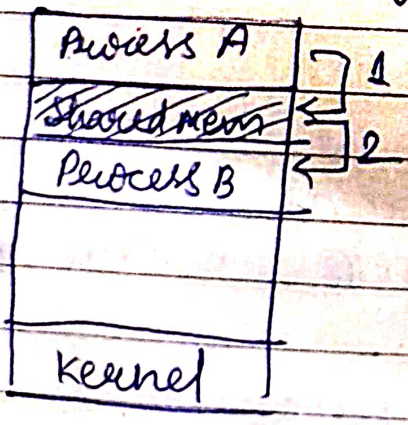
④ Process Termination
→ end of execution

Date: / / Page no:

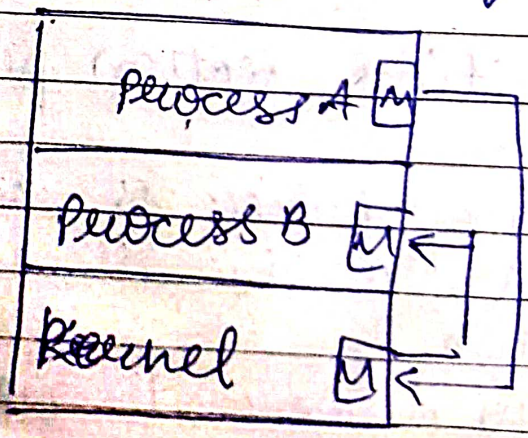
cooperating process.
→ process that affects / gets affected by another process
→ they might be sharing info
Adv
① data sharing
② ↑↑ computation speed
③ convinient
Methods of cooperation
→ communication → Sharing

Interprocess communication (IPC)
→ needed in cooperating process for exchanging info and data.
→ 2 fundamental models
↳ Shared memory
↳ Message passing

Shared Memory



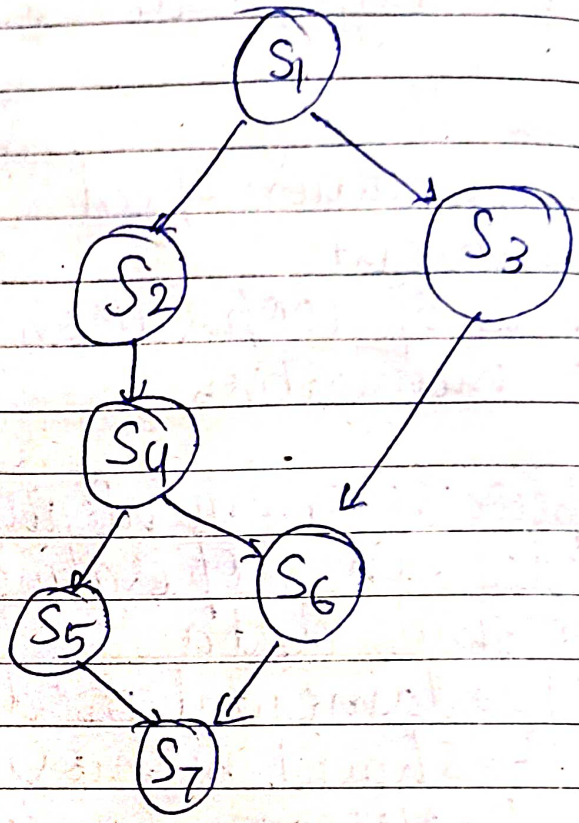
Message Passing



Date: / / Page no:

Precedence graphs
-> directed acyclic graphs where node corresponds to individual statement

```
S1;
count1 := 2
FORK L1;
S2;
S4;
count2 := 2
FORK L2;
S5;
goto L3;
L1 : S3;
L2 : JOIN count1;
     S6;
L3 : JOIN count2;
     S7
```



Critical section problem

- If one process is executing in critical section, no other process is allowed to execute in critical section
→ can only execute more than one process
→ design to ensure that race condition among processes will never arise.
→ must provide mutual exclusion
→ one process can't interrupt other processes in entering into critical section
→ must predict bounded waiting time
→ architectural Neutral

Date: / / Page no:

Semaphores

→ int variables that solves critical section problem
→ makes uses of wait and signal for synchronisation

Binary semaphor

wait ()
→ decreases value of arg if its +ve
→ else no opertion is performed

signal ()
→ increments value of arg S

```
wait(S) {
    while(S<=0); ——→ S is stuck untill S!=0
    S--;
}
signal(S) {
    S++;
}
```

Implementation:

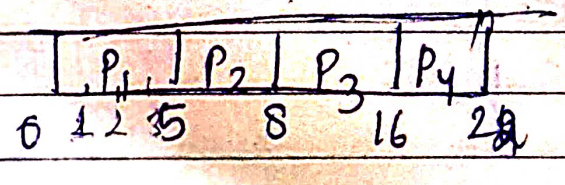
```
struct semaphore {
    enum values (0, 1);
    Queue < process > q;
}
P(semaphore S) {
    if (S.value == 1) {
        S.value = 0;
    }
    else {
        push(P);
        sleep();
    }
}
```

Date: / / Page no:

```
V(semaphore S) {
  if (s.value == 0
  if (q.q is empty) {
      S.value = 1;
  }
  else {
      q.pop();
      wakeup();
  }
}
```

Threads

Process	AT	BT	CT	TAT	WT
P_1	0	5	5	5	0
P_2	1	3	8	7	4
P_3	2	8	16	14	6
P_4	3	6	22	19	13



$\frac{23}{2} = 10.5$

Date: / / Page no:

Threads
→ flow of execution of process code with its own prog counter, system registers & stack
→ Basic unit of CPU utilisation
→ can perform one task at a time
process can be single threaded / multithreaded

Threads	Processes
→ light weight	→ heavy weight
→ no switching	→ requires switching
→ shares same resources	→ each process has own resources
→ if one thread is blocked, other thread in same task can execute	→ if one process is blocked, no other process can execute.

CPU scheduling.
~~Scheduling criteria~~
→ When CPU becomes idle, OS need to select one of process in ready queue to be executed. ie. done by CPU scheduler (short-term scheduler)

condition for CPU scheduling
↳ running process → waiting (wait for I/O to terminate child process)
↳ running process → ready (due to interruption)
↳ waiting process → ready (when I/O completes)
↳ process terminates
thus new process must be executed

Date: / / Page no:

preemptive scheduling: At times it is necessary to run a certain task process ~~eve~~ as it is of higher ~~proc~~ priority, the other process running is sent to waiting state without being executed completely and another ~~proc~~ process is being executed.

non preemptive scheduling: one process can be executed if and only if another ~~proc~~ running process has terminated execution or switched to waiting state due to some I/O tasks need to be completed (FCFS)

Scheduling criteria.

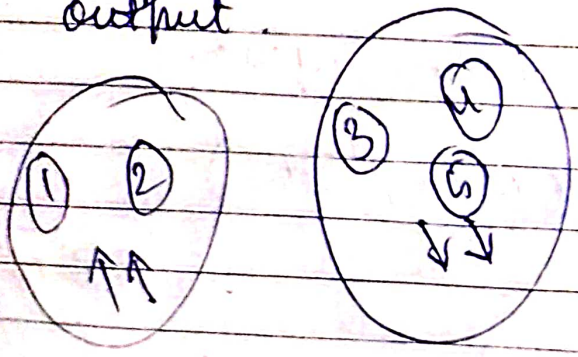
① → CPU utilisation: ↑↑ as possible (40-90%)

② → Throughput: no of process / time

③ → TAT: interval of time of submission to time of completion.

④ → WT: amt. of time spend by process in ready queue

⑤ → Response time: time taken to response to output.



Date: / / Page no:

Scheduling algo:

① FCFS (First come First serve)
→ maintained by FIFO queue.
→ when process enters ready queue, PCB is linked to tail of queue and is sent to idle CPU through its head.
→ non preemptive in nature

Process	AT	BT	CT	TAT	WT
A	3	4	7	4	0
B	5	3	13	8	5
C	0	2	2	2	0
D	5	1	14	9	8
E	4	3	10	6	3

Gantt chart

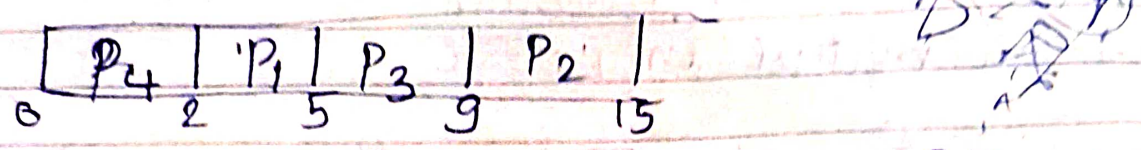
C	Idle	A	E	B	D
0	2	3	7	10	13	14

Disadvantage
→ convoy effect: smaller process had to wait for longer time

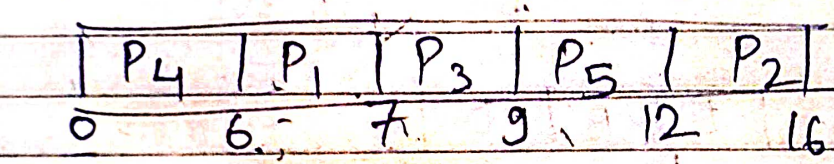
② SJF (Shortest Job first)
→ non preemptive in nature
→ if made preemptive then it is called SRTF (Shortest Remaining Time First)
→ if 2 processes have same burst time, then CPU decides according to FCFS algo

NON Preemptive SJF

Process	BT	WT	TAT
P_1	3	2	5
P_2	6	9	15
P_3	4	5	9
P_4	2	0	2



Process	AT	BT	CT	TAT	WT
P_1	3	1	7	4	3
P_2	1	4	16	15	11
P_3	~~2~~	2	9	5	3
~~P_4~~	~~0~~	6	6	6	0
P_5	2	3	12	10	7



Adv.
→ minimum avg. waiting time
→ Better avg. response as compared to FCFS

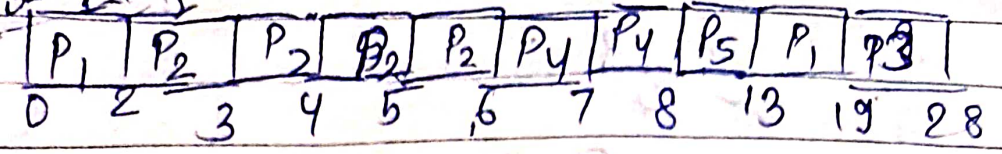
Disadv.
→ Process with TT burst time goes in starvation

~~SR~~ SRTF

Process	AT	BT	
P_1	0	8	~~8~~ 26
~~P_2~~	2	4	~~4~~ ~~8~~ 21
P_3	4	9	9
P_4	6	2	2
P_5	8	5	~~5~~ 4

Date: / / Page no:

Gantt chart.



P	AT	BT	TAT	WT	CT	TAT	WT
P_1	0	8	19		19	19	11
P_2	2	4	6		6	4	0
P_3	4	9	28		28	24	15
P_4	6	2	8		8	2	0
P_5	8	5	13		13	5	0

$\frac{54}{5} = 10.8$ $\frac{26}{5} = 5.2$

Round Robin Algorithm:
→ design for time sharing system
→ completing one system isn't imp
→ same as FCFS but preemptive
→ time quantum: unit of time in which processes time period is sliced
→ preemptive in nature

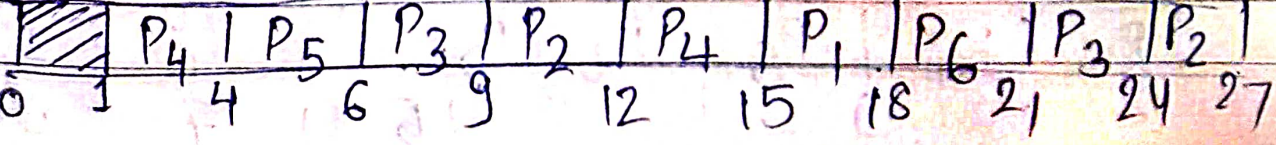
TQ = 3. Queue:

Process	AT	BT
P_1	5	2
P_2	4	3
P_3	3	1
P_4	1	6
P_5	2	2
P_6	6	3

P_4 P_5 P_3 P_2 P_1 P_6

P_4	P_1	P_3
27	30	32	33

gantt chart

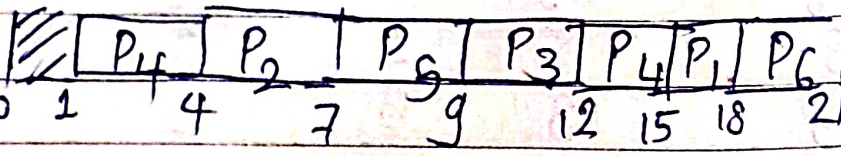


TQ=3.

Date: / / Page no:

Process	AT	BT
P_1	5	~~5~~ 2
P_2	4	~~6~~ 3
P_3	3	~~7~~ 4
P_4	1	~~9~~ ~~6~~ 3
P_5	2	~~2~~
P_6	6	~~3~~

~~P_5~~ ~~P_3~~ P_4 ~~P_2~~ ~~P_1~~ ~~P_6~~



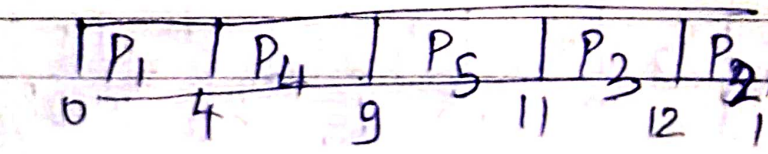
Priority Scheduling

- can be preemptive / non preemptive
- each process is allocated with priority
- process with higher priority possess first execution
- in case of same priority, FCFS method is executed

~~Non preemptive~~

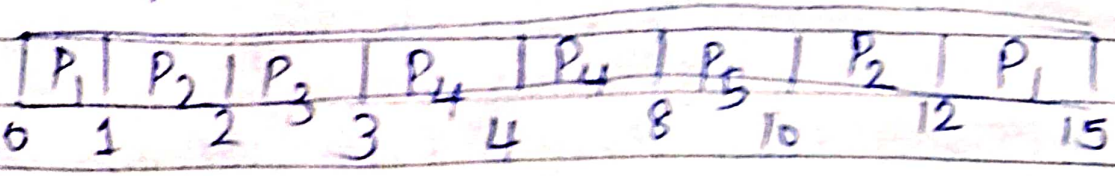
Pno.	AT	BT	Priority	
P_1	0	4	2	~~4~~ 3
P_2	1	3	3	~~3~~ 2
P_3	2	1	4	1
P_4	3	5	5	~~5~~ 4
P_5	4	2	5	2

non preemptive



CT	TAT	WT
~~4~~ 15	4	0
15	14	8
12	10	9
9	6	1
11	7	5

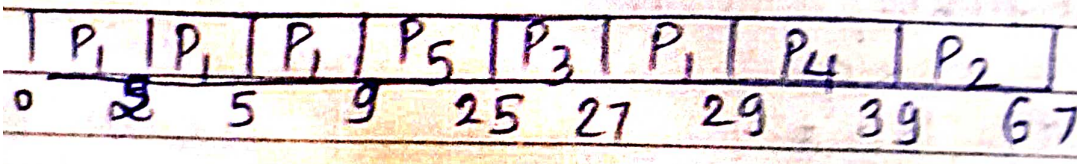
Preemptive.



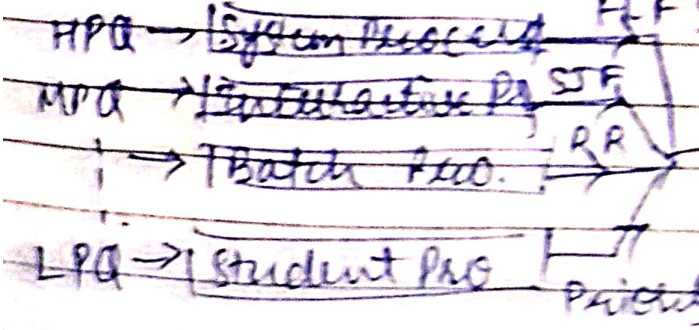
CT	TAT	WT
15	15	11
12	11	8
3	1	0
8	5	0
10	6	4

Preemptive

P	Process	AT	BT	CT	TAT	WT	
2	P_1	0	11	29	29	18	~~18 61~~
0	P_2	5	28	67	62	34	28
3	P_3	12	2	27	15	13	~~2~~
1	P_4	2	10	39	37	27	~~10~~
4	P_5	9	16 7	25	16	0	~~16~~



Multilevel Queue Scheduling
→ multiple Queues
→ Same priority = Same queue
→ each queue has its own algo



Multiprocessor scheduling

→ If multiple CPUs are available, load sharing becomes possible

→ While process scheduling is more complex as compared to single processor

→ In the cases when process are homogeneous

→ Thus can use any processor

Approaches to multiple processor Scheduling

(AP1) Master server: scheduling decisions
I/O processing.

Rest processors: user code.

↳ Reduces data sharing thus

↳ called Asymmetric multiprocessing

(AP2) ↳ each processor is self scheduling

↳ scheduling proceeds by having scheduler in each processor

↳ Symmetric multiprocessing.

① Process Affinity - soft, hard

② Load Balancing - push migration, pull migration

③ Multicore Processors - coarse grained Multithreading, Fine grained multithreading

④ Virtualisation and Threading

Date: / / Page no:

Deadlock → situation when each process is holding resource and waiting for another resource acquired by some other resource.
↳ set of processes are blocked

cause of deadlock
→ mutual exclusion → Hold & wait → No preemption → Circular wait

Concepts of Memory Mgmt:

→ every process need memory for storing variable and code of instructions

→ Mgmt of memory is required when there are more than one process at a time.

→ mgmt avoids reading/damaging of processes' memory by each other

Functions

↳ keeping memory address ↳ determine allocation policy

↳ deallocation technique ↳ allocation technique

Requirements

↳ Relocation ↳ Protection ↳ sharing

↳ Logical organisation ↳ physical organisation

Address Binding

→ allocates physical memory location to logical pointer by associating physical address to logical address

Steps of binding

↳ compiler time: absolute code is generated at this time

in case memory location changes, it is necessary that code must be recompiled

↳ Load time: compiler must generate relocatable code after compile time.

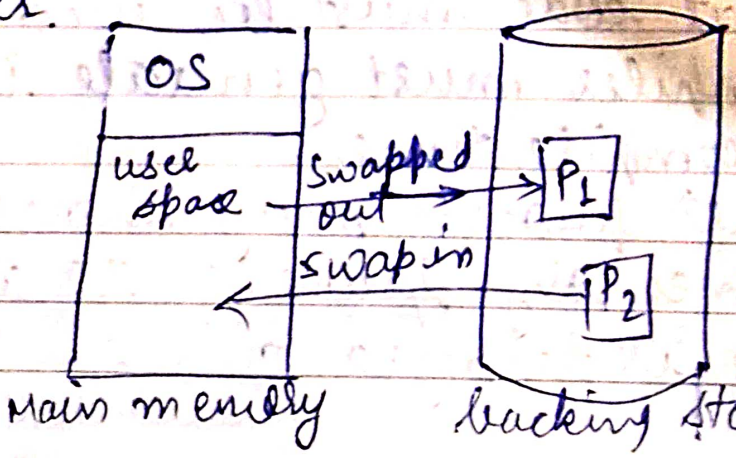
↳ execution time: process may move from one memory process to another thus binding is delayed till runtime.

Logical address
→ generated by CPU
→ referred as virtual address
→ LAS (logical address space): set of all LA generated by program

* MMU: mapping of virtual & physical address at runtime is done by it (Memory mgmt unit)
* Relocation register: Base Register (added to each and every address)

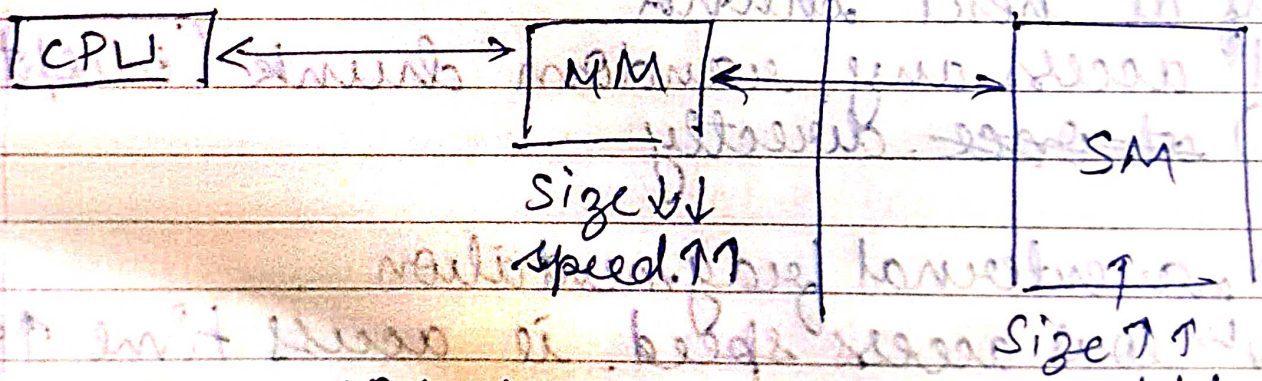
Physical address
→ address of main memory and actually loaded into Memory address Register
→ PAS: set of all physical address corresponding to logical address
at compile time PA = LA but at execution time PA ≠ LA

Swapping
→ removing inactive process temporarily from memory
→ done to allocate this memory to other process
→ inactive process is copied to secondary mem
→ when it is swapped back, it's image (except abt) is copied into new block allocated by mem-manager.



dispatcher: when a process is being decided to be executed, it is used.
Ready queue: all processes whose memory images are on the backing store or in memory and are ready to run are placed.
→ address binding at load time: process is moved to same location of prev one
at execution time: can be addressed where it is computed during execution time.
→ well suited for time sharing os.

*contiguous and non contiguous memory
→ size of mem & access time.
→ thus we need 2 diff memory. Thus we have main memory & secondary



→ OS will decide which process will reside in which part of MM.
→ CPU generates LA(S.M) which is translated into PA(M.M) using for accessing main memory

Contiguous Memory
→ Process allocation from SM to MM is contiguous (क्रम के लिए क्रम) in nature.
→ whole process is fetched at once.

Advantages
↳ access time ↓↓

*
External fragmentation → problem arised when
→ we have enough space for allocation but unable to do so due to space contiguous fashion.

Adv ↳ access time ↓↓ ; ↳ address translation
Disadv ↳ External fragmentation

Non contiguous memory
↳ process is splitted into small chunks in order to store it in MM.
↳ eg: linked list
↳ data arranged anywhere ie first chunks points at next chunk
↳ can't access any random chunk (except first) directly.

Adv ↳ no external fragmentation
disadv ↳ low access speed. ie access time ↑↑

contiguous memory allocation

fixed size partition

variable size partition

Contiguous Memory allocation (techniques)

① Fixed size partitioning.
→ capacity is divided in fixed chunks.

Problem: Internal fragmentation
→ wastage of size due to fixed partitioning as we haven't designed partition according to process size.
→ one partition ≈ one process
→ one partition can't be reused.

② Variable sized partitioning (Dynamic).
→ no partition of capacities initially
→ no size of new chunk ≈ size of process.
→ no internal fragmentation

algos for contiguous memory allocation

~~First fit, first process~~

First fit:
→ follow FCFS for allocation of memory.
→ searches from left to right for empty block.
→ process are also chosen acc to FCFS also.
→ in variable sized partition, reqd. sized is allocated and rest is partitioned for rest other processes.

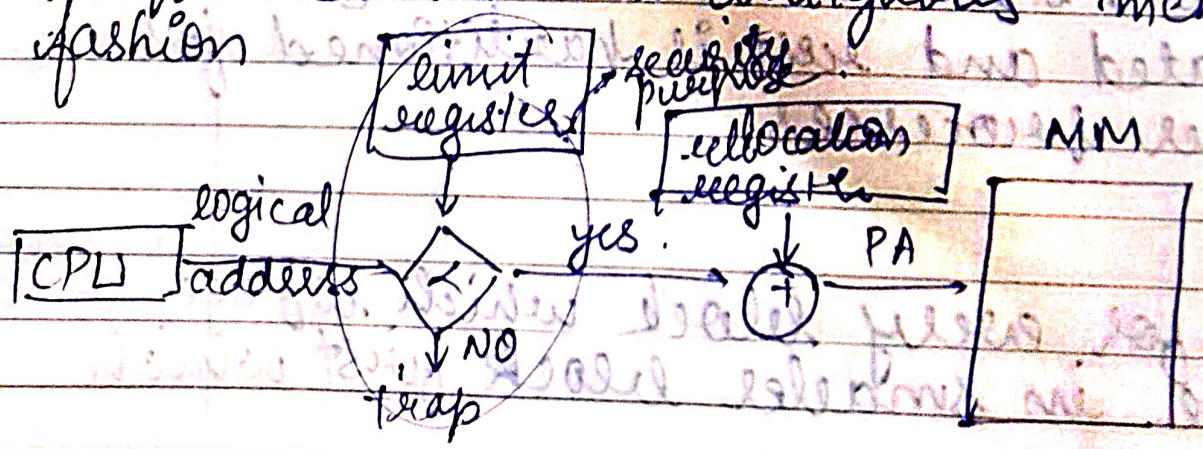
Best fit:
→ search for every block which is free
→ allocate in smaller block first which is free
→ process chosen again acc. to FCFS.

→ sometimes causes in external fragmentation
→ fixed size : it performs best
→ variable size : it performs worst.

worst fit :
→ allocate largest block first which is free.
→ actually works best in variable size partitioning.

* condition of external fragmentation :
size (process) > size (Available)
* Internal fragmentation = wasted in fixed sized partition
* External fragmentation = size of process (case when size (process) > size (chunk))

~~Paging~~
Address Translation
→ CPU generates ~~memory~~ address for SM only
→ It doesn't know about MM
→ Imp data is retrived by converted from LA to PA
→ needed as search time for SM is more due to its large size
→ thus we convert LA to PA for shorter access time
→ we pick SM which is contiguous memory fashion



allocation register : contains base address of

* limit register: limits access of data uptill the authorised access. used for security issues
* process accessing gets trapped if not passes condition.

* LA = PA + relocation register address
* PA < LR

Paging.
→ we switch to non contiguous memory allocation just to eliminate problem of external fragmentation
→ we'll partition SM as well as MM when we talk about non contiguous memory.
→ we will ~~pass~~ divide ~~each part~~ SM into equal sized partition that is called page

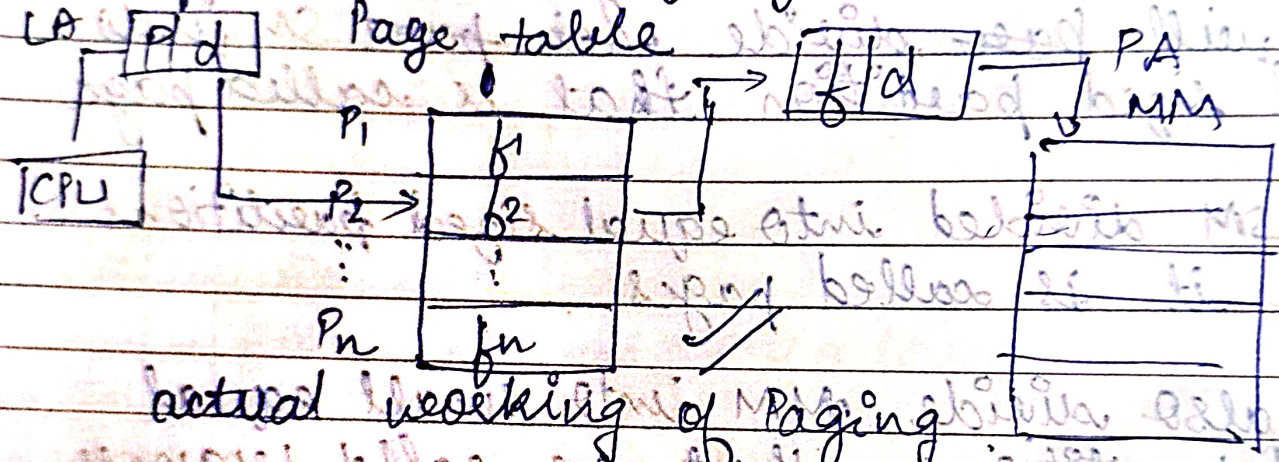
* pages: SM divided into equal sized partitions then it is called pages.

→ we also divide MM into small equal sized partitions that are called frames.

* frames: MM divided into equal sized partitions then parts are called frames
* frame size depends on page size.
  frame size = page size
→ whereever we find empty frame in MM, we just fill the space with our process divided in pages.
→ done for avoiding external fragmentation
→ we've divided process into pages

Address translation in Paging.
→ CPU generates LA
→ LA divided into → | p | d |
p = page no. d = Instruction offset.

→CPU creates such type of LA for non contiguous memory automatically.
→ frames in MM contains Pages similar to linked list ie. address of next page is stored as pointer in previous page.
→ we have base address initially
→ if we do this then we suffered from very slow access
for this disadvantage.
→ we maintain data structure called Page Table contain no. of entries
no of entries = no of pages in SM



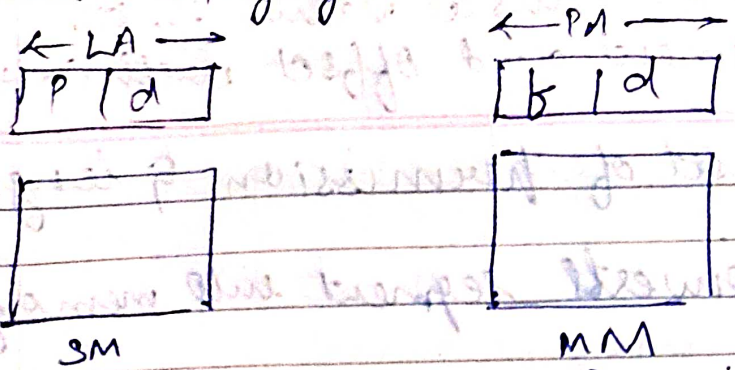
actual working of Paging

Advantages ↳ fast access ↳ no external fragmentation
Disadvantage ↳ we may suffer from internal fragmentation

size of address → no of address required
if we have n bit address and size of each chunk is 1B (generally)
then
size of memory = $2^n \times 1B$

$2^{10} = 1K$ $2^{20} = 1M$ $2^{30} = 1G$ $2^{40} = 1T$ $2^{50} = 1P$

Numerical on paging.



LA = 24 B PA = 16 B PS = 1 KB

Size of memory = $2^4 \times 2^{20} \times 1B$

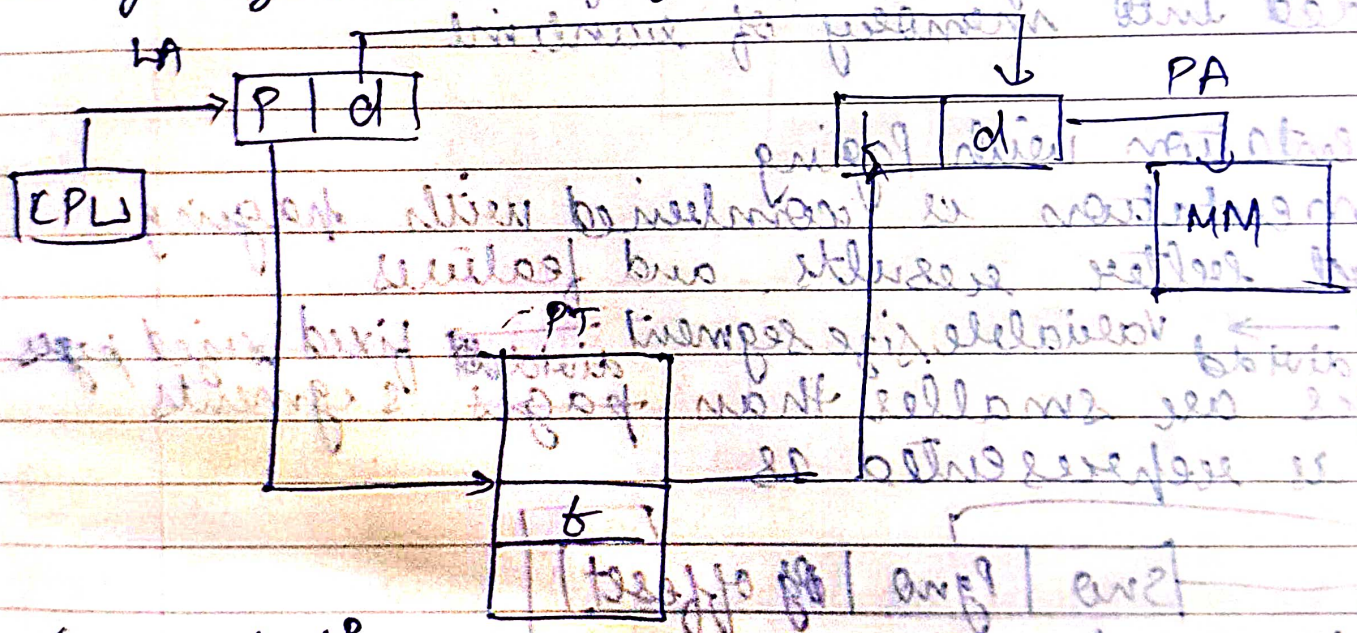
= 16 MB

size of MM = $2^6 \times 2^{10} \times 1B$

= 32 KB

Page size = $\frac{1KB}{1B}$ = 1024 locations = 2^{10}

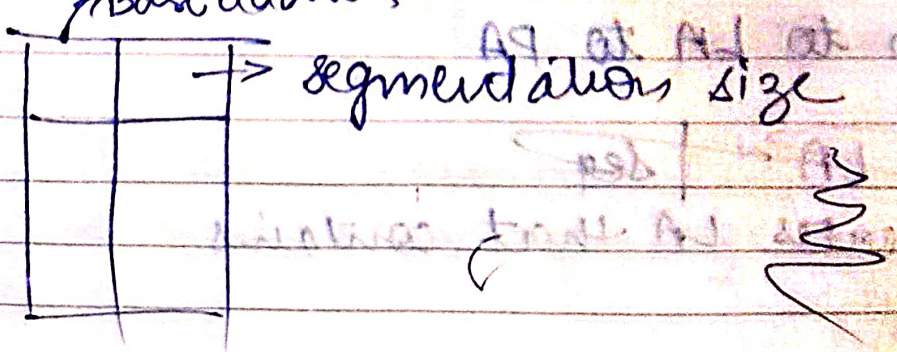
Page size will always = Frame size



Segmentation

→ A program is a collection of segments.

→ most common way to achieve memory protection



→ Instruction Operand contains a value that determines a segment and offset within that segment

→ a segment has a set of permission & length associated with.

– memory mgmt converts segment into memory address

Types of segmentation

① Virtual Memory segment: each process is divided into no. of segments, not all of which ~~is don't~~ resident any pt. of time.

② simple segment: each process is divided into no. of segments all of which are loaded into memory of execution

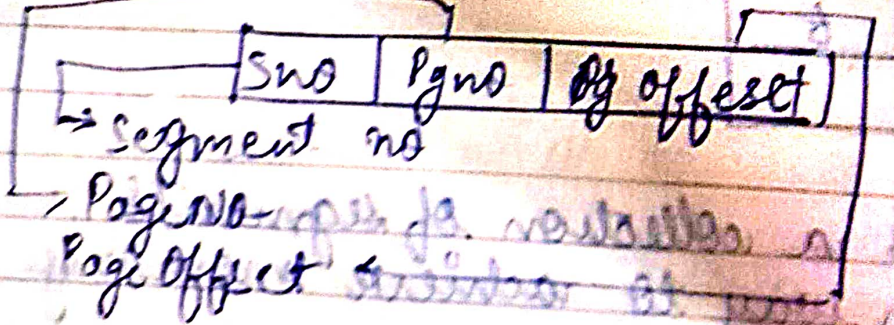
Segmentation with Paging

– Segmentation is combined with paging to get better results and features

→ MM divided variable size segment divided fixed sized pages

→ pages are smaller than segments

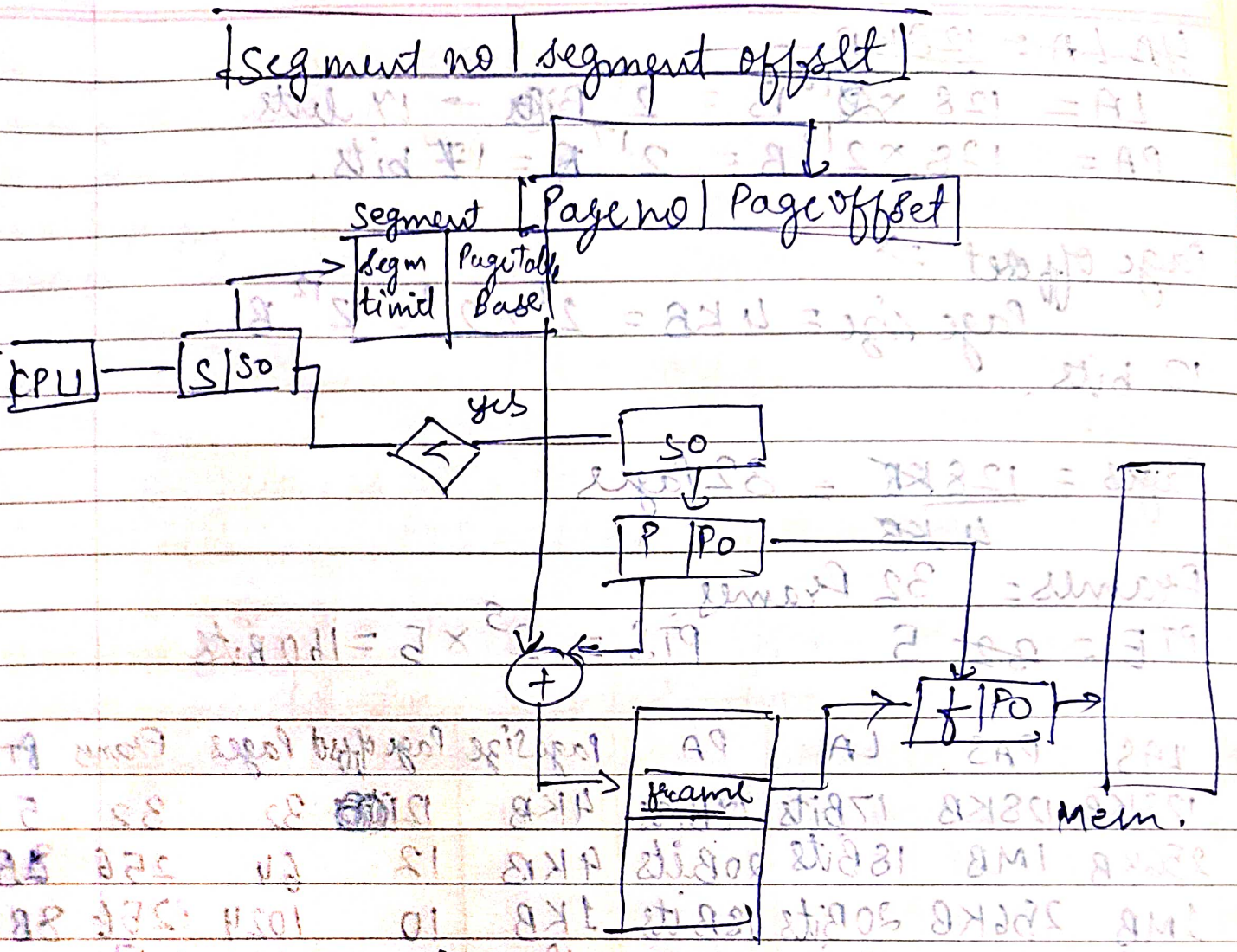
→ LA is represented as



Translation to LA to PA

CPU → LA – | seg

→ CPU generates LA that contains



~~Multilevel Paging~~

Paging Numericals.

→ LAS : size of process
→ PAS : size of MM
→ LA : No of bits reqd. to represent LAS
→ PA : No of bits reqd. to represent PAS.
→ Pages : LAS divided in eq. blocks
→ Frames : PAS (MM) divided in eq blocks
→ Pagesize : size of Partition = Frame size
→ Page offset : No. of bits rep. particular byte in page
→ PTE : identify frame no.
→ PTS : no. of entries in page table

Q. LAS = 128 KB PAS = 128 KB

Frame size = Page size = 4 KB.

~~LA : 128 KB~~

$LA = 128 \times 2^{10} B = 2^{17} B$ → 17 bits.

$PA = 128 \times 2^{10} B = 2^{17} B$ = 17 bits.

Page Offset ⇒

Page size = $4 KB = 2^2 \times 2^{10} = 2^{12} B$

12 bits.

$Pages = \frac{128 KB}{4 KB} = 32$ Pages

Frames = 32 Frames

PTE = ~~32~~ 5 PTS = $2^5 \times 5 = 160$ Bits

LAS	PAS	LA	PA	PageSize	Page offset	Pages	Frames	PTE
128KB	128KB	17Bits	17Bits	4KB	12	32	32	5 B
256KB	1MB	18Bits	20Bits	4KB	12	64	256	8B
1MB	256KB	20Bits	18Bits	1KB	10	1024	3256	8B
1MB	512MB	20Bits	29Bits	2^{12}	12	256	2^{17}	17B
4MB	2MB	22Bits	21Bits	2^{12}	12	2^{10}	2^9	9B
4MB	4MB	22Bits	22Bits	2^{14}	14	2^8	2^8	

$LA = \log_2(LAS)$ Page offset = $\log_2$(Page size)

$PA = \log_2(PAS)$ Pages = $\frac{LAS}{Page size}$

Frames = $\frac{PAS}{Page Size (Frame)}$ PTE

$PTE = \log_2$(frames)

PTS = No. of pages * size of each entry in PTE

(for Byte addressable)

Multilevel Paging

- more than one page table

LA = 22 bits thus LAS = 2^{22} B

Page Size = 4KB offset = 2^{22} B = 12 bits

Page no = $\log_2\left(\frac{LAS}{Page\ Size}\right)$ = $\log_2 LAS - \log_2 Page\ Size$

= LA − offset

= 22 − 12 = 10.

Pages in process = 2^{10} = 1K pages

PTE = 4B PTS = (1K * 4) B = 4KB

Working Set: Portion of SM in MM that could be referred (time period: principal of locality

PTS

20B → Page table will be in main

64B memory

1024B

544B

1152B

256B

concept of multilevel paging

Q. LAS = 32 Bits (process size = 2^{32})

page size = 4KB = Frame size

No of pages = $\frac{2^{32}}{2^{12}} = 2^{20}$ pages

PAS (MM) = 2^{44} B (Not Possible but we'd assume)

No. of frames = $\frac{2^{44}}{2^{12}} = 2^{32}$ Frames

PA = 32 Bits (bits req·d in one frame)

MM and PT doesn't contain Page no. but there is array that hold page no. of PT outside

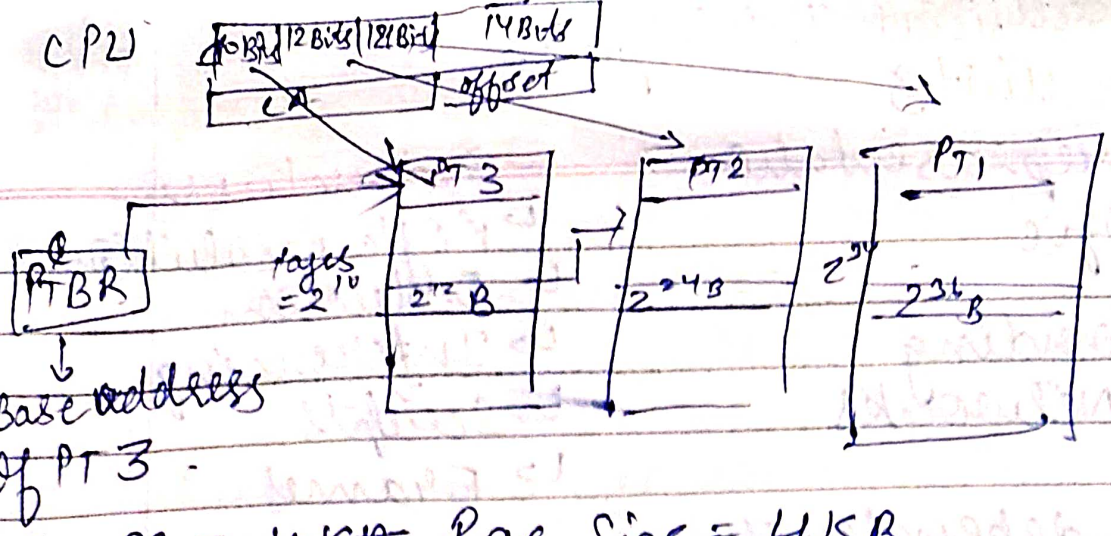
PTS = no. Of pages × bits in frame
= $2^{20} \times 32$ bits
= $2^{20} \times 4$ bytes
= 4MB

NE

Q. given — Required data
LA = 48 bits — PT1, PT2, PT3, Address Split
Page size = 16KB
PTE = 4B

LA = 48b ⇒ LAS = 2^{48} B
Page Size = 16KB = 2^{14} B
No. Of pages = $\frac{2^{48}}{2^{14}} = 2^{34}$ Pages

Page table level 1.
No. Of Pages = 2^{34} PTS = $2^{34} \times 2^2 = 2^{36}$ B
Since
$2^{36} > 2^{14}$ or PTS > Page size
Thus we go for another level paging
Page table level 2
No. of pages in PT2 = $\frac{2^{36}}{2^{14}} = 2^{22}$ pages
PT2S = $2^{22} \times 2^2 = 2^{24} > 2^{14}$
Thus we go for another level paging
Page table level 3
No. Of pages = $\frac{2^{24}}{2^{14}} = 2^{10}$ pages
PT3S = $2^{10} * 2^2 = 2^{12} < 2^{14}$
Can fit in frame



Q. ~~PS = 4KB~~ Page Size = 4KB
PTS = 4KB Level of Paging = 1.
PTE = 4B.
LAS = ?

PT = 4KB Page size = 4KB
Thus level of paging = 1 as PT = Page Size.
PTS = 4KB PTE = 4B.
No. of Pages = $\frac{4KB}{4B}$ = 1K pages.
LAS =
= 1K × 4KB = 4×2^{20} B = 4MB //

~~Segmentation~~

Unit 4 (OS)

Virtual Memory (VAS = Virtual Address Space)
→ not actual memory.
→ adv: process size > main memory, OS helps in executing process
→ user gets illusion that MM is big enough to execute program.
→ process creation is done in SM.
→ VAS is limited to SM (being cheap)
→ process is divided in parts and MM executes all parts one by one.

Main advantage: Degree of multiprogramming ↑↑.

Virtual Memory

↳ not actual memory ↳ ~~if process~~

↳ if size(process) > size(~~RAM~~ MM.), OS helps in execution using this concept (makes user an illusion)

↳ main advantage: Degree of multiprogram-ming is pages.

↳ process are arranged in MM non-contiguously

↳ we keep 1 bit in Page table that tells whether process is present/not in MM.

→ if page we are searching is not present in MM, then it results in page fault.

Page table attributes

Pno	Fno	P/A	D	R	P

Pno: Process no
Fno: Frame no
P/A: present/absent in MM
D: Dirty Bit (updations/modifications are done in process in MM)

Process with Dirty Bit = 1 must need to have changes in secondary memory as well. (this modification is called write back)

Reference Bit (R): those no of pages referred in last clock cycle. ~~are marked (1) Refer~~

Protection Bit (P): rights of access to the data.

*Demand Paging
→ don't load any page until required
Problem: page fault

* Page fault: Page you are searching in MM is actually not present in MM.

* Context switching: currently running process is stopped and OS request to start another process.

MMAT = ns CST (U → OS) = microsecond
HD to MM = ms. CST (OS → U) = microsecond

~~* Thrashing~~:
Thrashing: Read. when there are page faults
→ page faults ↓ → OS fetches page from SM and do context switching with Page in MM, if it happens at high rate, OS need to spend time for it. ~~This is called thrashing~~

* Locality of Reference :- only those pages are referred that takes part in lifetime of any running process
→ reduces the access time
→ takes time for first time then it does n't take much time.
→ stores pg no & fr. no after 1st access
→ makes use of cache mem.
→ entire TLB needs to be cleaned when we want to do context switching.
→ cache memory is faster than MM but cheaper than Register.

TLB is also called Associative memory
→ makes use of TAG for frame no. searching
→ we need to access any element for the first time in MM but after 1st access, it gets added in our TLB
→ thus we don't need to access MM again and again.

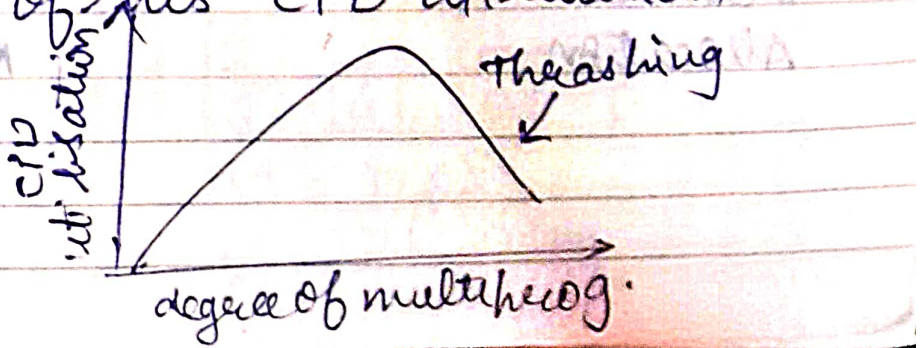
TLB Hit = (TLB + MM) TLB Miss = (TLB + PT + MM)

if TLB Hit = x% TLB miss = (1 - x)%

$$EMAT = \frac{x}{100}(TLB + MM) + \left(1 - \frac{x}{100}\right)(TLB + PT + MM)$$

Numerical

→ Such state is called Thrashing. Since because of this CPU utilisation will reduce



Solution to Thrashing is Page Replacement algo page fault amount & time read for memory acces

Numericals

Q. Page fault service time = 10ms
avg memory access time = 20 ns.
one page fault generated after every 10^6 memory access.

$$EMAS = p * (\text{page fault service time}) + (1-p) * (MAT)$$

$$= \frac{1}{10^6} (10) * 10^6 + \left[\left(1 - \frac{1}{10^6}\right) * 20\right]$$

$$= 30 ns \text{ (approx)}$$

Demand Paging
↳ it becomes difficult to decide whether any part of process (page) that is kept in MM could be useful or not.
↳ Thus to overcome this problem we introduced demand paging
↳ It suggests to keep all pages of frame in SM until they are required.

Page replacement and Frame allocation

Virtual Memory Implementation

- Frame allocation
- Page Replacement

Frame Allocation
- deciding which process will get how many frames to fill pages into them.
min limit: no of pages holded by instruction (1 for code, 2 for data, 1 for stack)
max limit: whole process

FA

equal allocation
- no. of frames must be divided by no. of process
- limited: some process might load as a whole
- unequality
- weights not considered.

weighted allocation
↳ determines no. of pages in each process
↳ ~~find percent~~ weightage of each process
$\frac{P_1}{\sum P_i} \times$ frames total
$\frac{P_2}{\sum P_i} \times$ frames

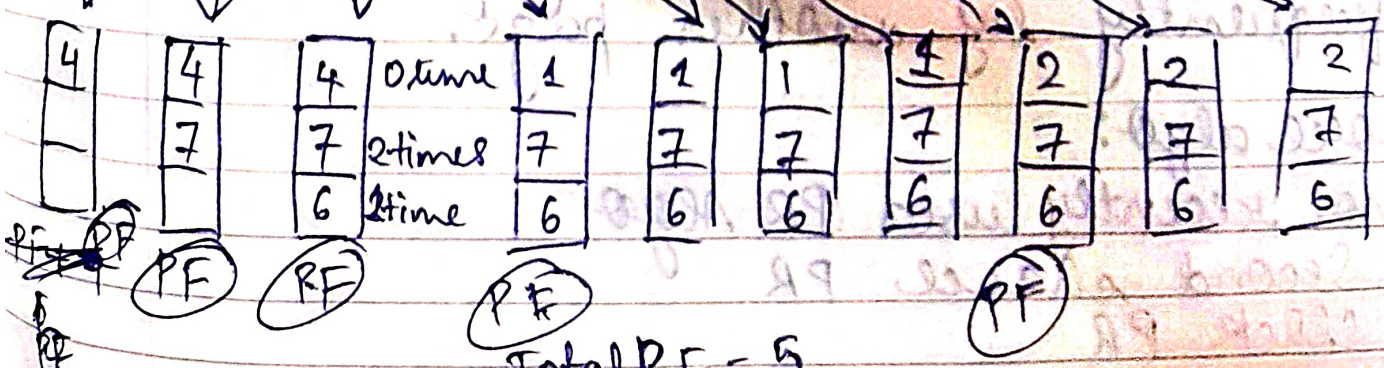
Dynamic Allocation
↳ allocates frames dynamically on basis of priority
↳ strictly based on priority

Page Replacement Algorithm

1) Optimal
→ that page needs to be replaced that doesn't has to be refered much in future ↳ problem is we can't predict futures at times
→ considers future references

4, 7, 6, 1, 7, 6, 1, 2, 7, 2

F = 3



Total PF = 5

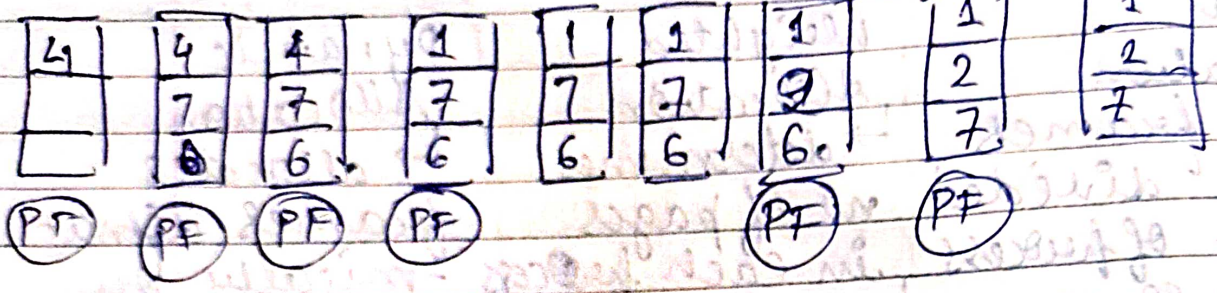
~~2) LRU~~

~~4, 7, 6, 1,~~

2) FIFO (First in first out)

4, 7, 6, 1, 7, 6, 1, 2, 7, 2 (F = 3)

→ we need a queue. queue [4, 7, 6, 1, 2, 7]



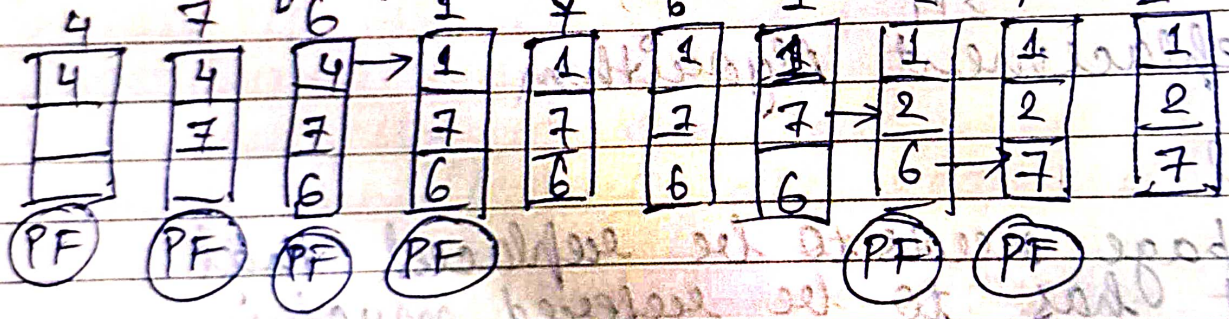
total Page Faults = 6.

3) LRU (Least recently used)

4, 7, 6, 1, 7, 6, 1, 2, 7, 2 (F = 3)

→ see the past data.

→ least frequently added is prefered.



total PF = 6.

Huge disadvantage: At times new page that has to be accessed from MM might have been replaced very just frequently by another page.

More algo:

→ not recently used PR Algo

→ Second chance PR

→ CLOCK PR.

Types of page replacement

Local → even though space (MM) is free, then we'd replace page

Local → if all the frames allocated to a process are filled and one more page needs to be added to the frame, then we'd need to replace any of the page from the allocated frame with new page irrespective that there is more space in MM. ↑ [Preferred]

Global → New page can be replaced by any least priority page in MM ~~irrespective of fact that we can only allocate frame~~ it is not necessary that we need to allocate ~~frame~~ only those frames that are allocated to process

Page Replacement algo numerical

0100 0200 0430 0499 0510 0530 0560

0120 0220 0240 0260 0320 0370

each page contains 100 records thus, pages:
1 2 4 4 5 5 5 1 2 2 2 3 3
we have only one memory frame.

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

[1] [2] [4] [4] [5] [1] [2] [3]

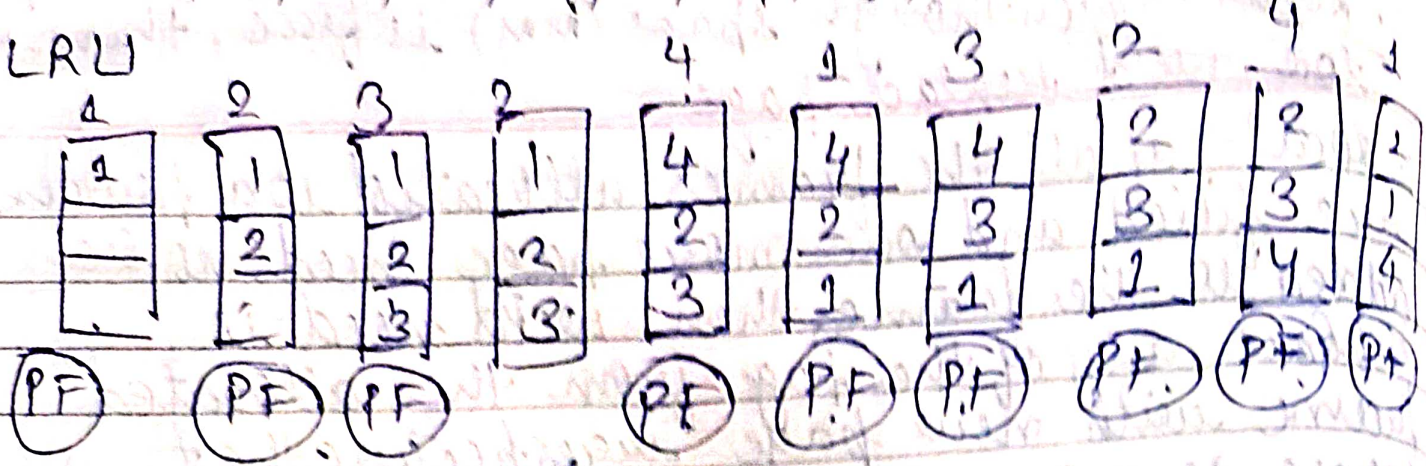
PF PF PF PF PF PF PF

no. of PF = 7.

Hit ratio = $\frac{6}{13} = \frac{TP - PF}{TP}$ Miss ratio = $\frac{7}{13} = \frac{PF}{\text{Total Pages}}$

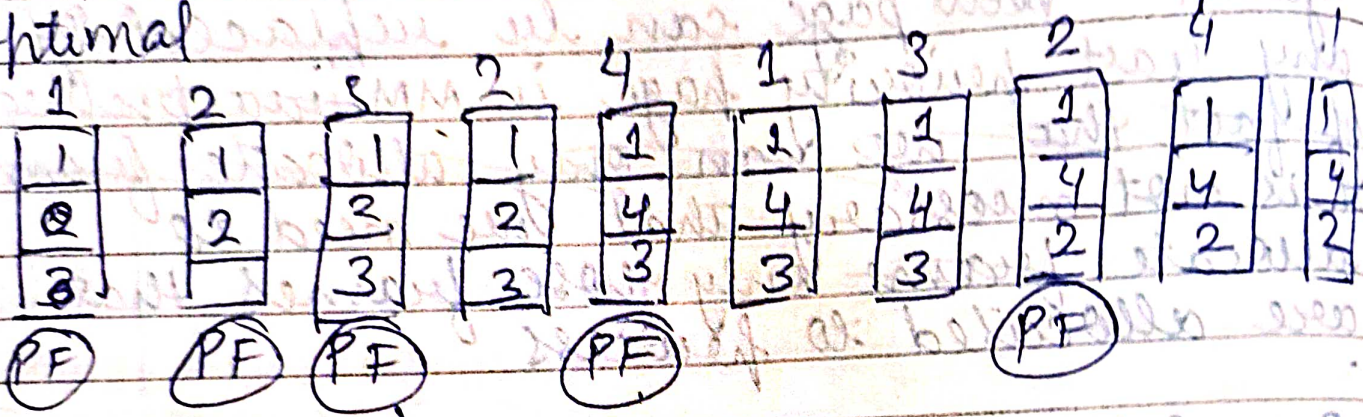
Q. 1, 2, 3, 2, 4, 1, 3, 2, 4, 1

LRU



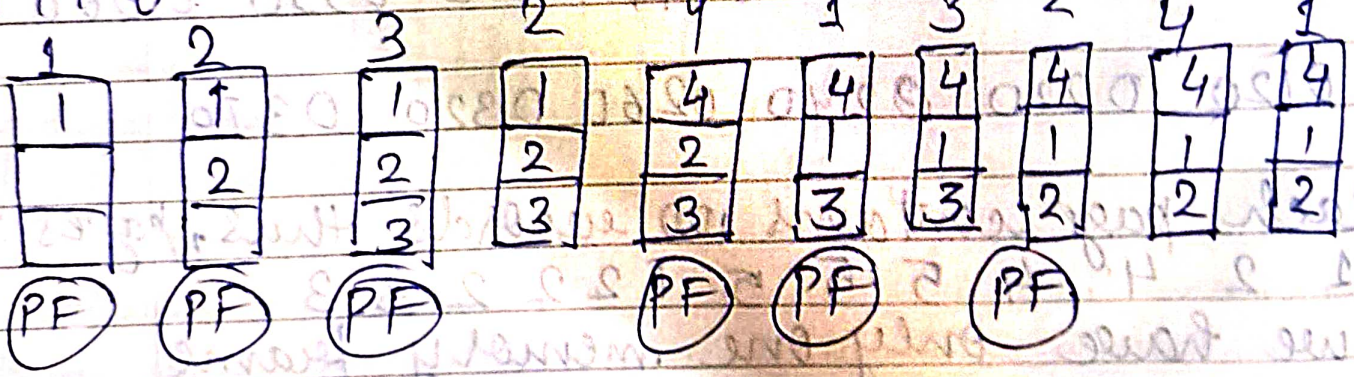
PF(LRU) = 9/10

Optimal



PF(optimal) = 5/10 ~~1 2 3 4 1 2~~

FIFO



PF(Fifo) = 6/10.

(b) Optimal < FIFO ≤ LRU

Q. A memory page contains a heavily used variable that was initialised very early and is in constant use is removed when — is used.

1) LRU ☐ 2) FIFO ☑

3) LFU ☐ 4) None ☐

Q. → system uses FIFO policy
→ 4 page frames with no page loaded
→ access pages (distinct) and then same in reverse order
→ no. of Page faults?

a) 196 ☑ b) 192 ☐ c) 197 ☐ d) 195 ☐

Sol. 100 + 100 − (4) = 200 − 4 = 196.
↳ Pages hit

Belady Anamoly
→ no of frames allocated when increase also increases the no. of page faults occuring
→ This unexpected condition is called Belady's Anamoly
→ occurs in FIFO

Stack Property
→ identify whether Belady Anamoly will occur or not.
→ page available in method with lower no. of frames but not in method with higher no. of frames

Demand segmentation

→ similar to demand Paging
→ used when there is insufficient hardware available to implement demand paging.
→ There is valid Bit in segment table that specify presence of segment in physical mem.
→ absence of segment in MM results in segment fault.
→ allows pages (referenced to each other) to be brought in memory together to decrease no. of page faults.

Role of OS in security

→ security mgmt of an OS helps in implementing mechanisms that secure and protect the computer system internally and externally
→ OS must protect itself from security breaches such as runaway processes memory access violations, stack overflow violations, the launching of programs with excessive privileges etc.

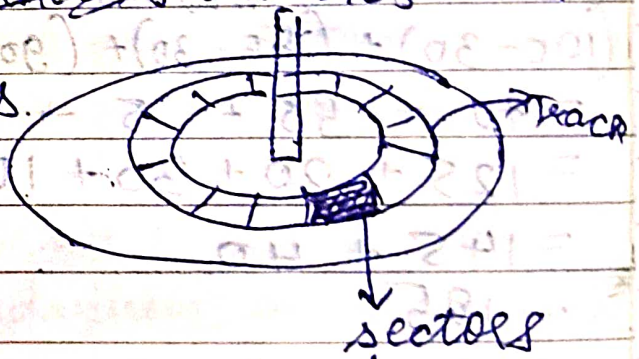
Security Breaches

→ occurs when an intruder gains unauthorised access to an organisation protected system and data.

Disk schedule.
- we need to perform read/write op^{n} of data present in disk
- outcomes are possible :- ① success, ② failure, ③ partial success
- The techniques that OS uses to determine the request which is needed to be satisfied is called disk scheduling.
- It is required to maintain efficiency and speed of process execution

Terminology
- Platter :- cylindrical disks present at disk
- Spindle :- spins platter
- read write head :- performs R/W operations.
- No of read write head = 2 * No. of platter
- Tracks :- ~~disk~~ platter is divided into tracks ~~of~~ with concentric circles
- Sectors :- ~~platter~~ track divided into small parts.
- cylinders :- logical view of all ~~cylinders~~ tracks with same radius

HD → platters → tracks → sectors.
collection of, collection of, collection of



* seek time :- time taken by magnetic heads to reach desired ~~segment~~ track from its initial position
* latency time :- time taken by mag. head to reach desired sector from its initial position
* Allotment unit = ST + LT
* Density :- No. of bytes in unit track.
* Transfer time = Read Time + access Time.
  (also depends on speed of rotation of disk & no. of bytes transferable)

Disk access Time = ST + TT + LT.

Bad sectors: Sector that can't decide whether to read 0 or 1

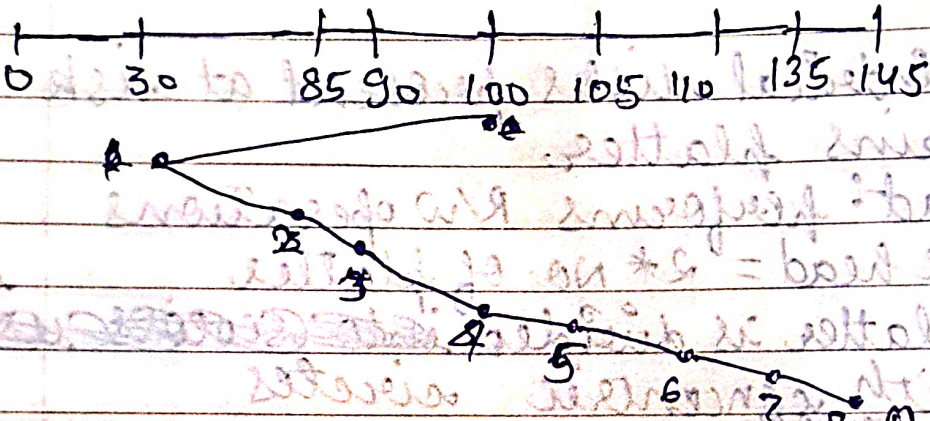
Disk Scheduling algo.

1) FCFS

→ first cylinder which comes first will get read/write op^{n} first.

Q. Disk has 201 cylinders. Initial posi^{n} of r/w head = 100 ↳ queue of Disk access req:

30 85 90 105 110 135 145



Here 90 is accessed after 2 req. (30 then 85)

Total arm movement:

(100-30) + (85-30) + (90-85) + (105-90) + (110-105) + (135-110) + (145-

= 70 + 55 + 5 + 15 + 5 + 25 + 10

= 125 + 20 + 30 + 10

= 145 + 40

= 185

Adv

→ each & every req. gets chance for execution

→ indefinite postponement ✗✗

Disadv.

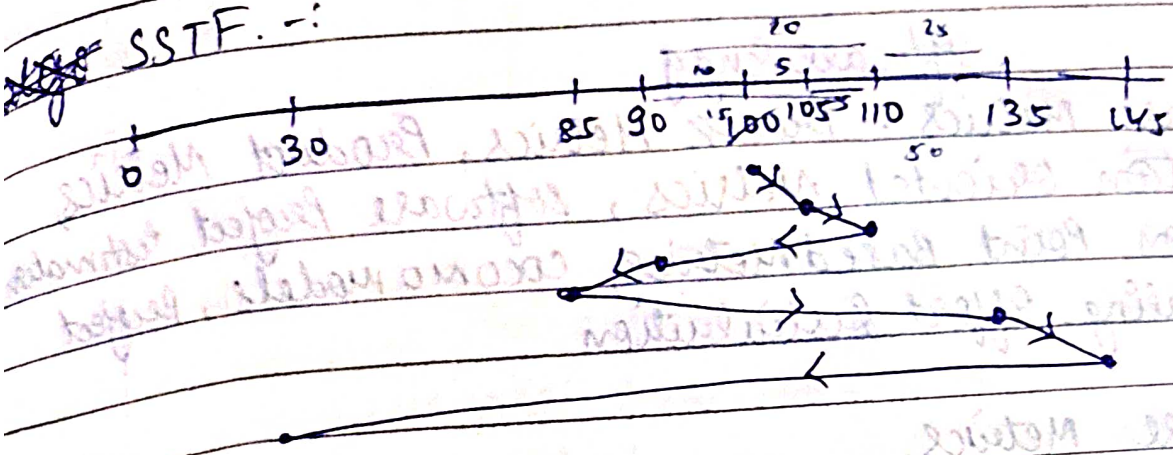
→ can't optimize STORAM

→ doesn't give good performance at times.

2) SSTF Disk scheduling (shortest seek time first)
seek time calculated in advance
response time ↓↓ & throughput ↑↑

same Q.
30 85 90 105 110 135 145

SSTF :-



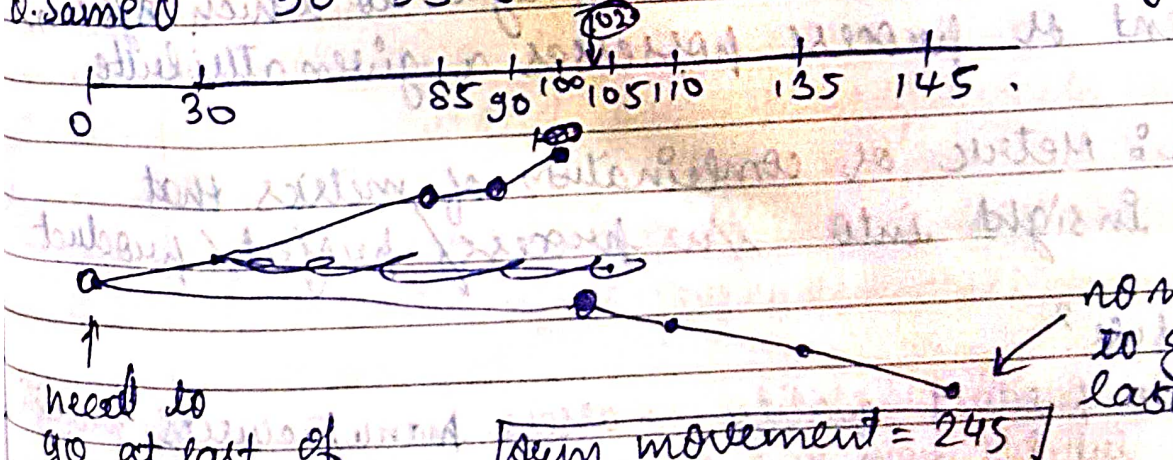
~~Total arm mov~~
90 is serviced after 2 requests (105 then 110)
arm movement
∴ (110-100) + (110-90) + (90-85) + (135-85) + (145-135)
+ (145-30)
= 10+20+5+50+10+115 = 130+55+125
= 30+180 = 210

Adv
→ Avg response time (mostly) ↓↓
→ system throughput ↑↑

Disadv
↳ Seek time (in adv) its overhead
↳ May occur starvation

3) Scan disk scheduling (elevator).
→ goes unidirectional first and then reverses order for executing rest disks.

Q. Same Q 30 85 90 105 110 135 145 (previous req. was at cylinder 102)



Arm movement = 245