OS

→ Job / Punch card OS (Job = Program + Ctrl info + I/P data)
→ Batch (Batch same type of instructions)
→ Spooling (Store instructions of a process/task in mem.) (I/O Peripheral Operations Online or Jobs)
→ Multiprogramming (make a Pool of Processes who wants CPU) (keeps CPU busy)
→ Multitasking (All tasks in pool execute) (Time sharing)
(Multiprogramming with Round Robin) (illusion of Parallelism)
→ Multiprocessing (more than 1 CPU)
→ symmetric
→ Asymmetric (master-slave)

→ Real time
→ Soft RT (t=0+)
→ Hard RT (t=0)
→ Firm RT

→ Distributed OS (different nodes connected by n/w)
→ Resource sharing (Anydesk)
→ Computation speedup (Bitcoin mining)
→ Reliability
→ communication

*) System call
*) mode bit
0 → kernel
1 → user

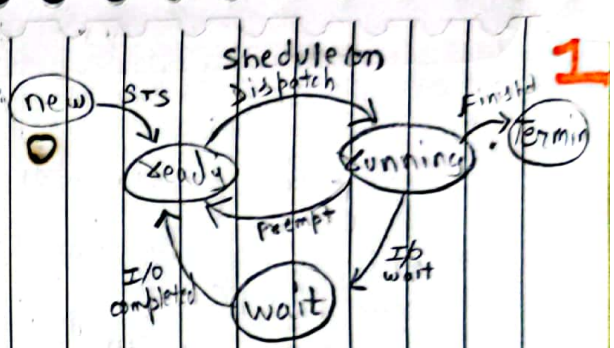
*) user & OS Interface
• Command Interpreter
• GUI

*) Structure of OS:
→ Simple structure
→ Layered approach
→ Micro kernel approach

*) Process = Program in execution
→ Text section (code)
→ Stack (Local vars)
→ Data section (Global var)
→ Heap (Dynamically allocated)

*) PCB (way to represent process in OS)
• Process state
• Program counter
• CPU Register
• CPU scheduling information
• Memory management information
• Accounting info.
• I/O status ,,

*) Process States
① New (Process created)
② Running (Executing instr.)
③ waiting (event wait)
④ Ready (Process waiting to be assigned)
⑤ Terminated (process over)



note: Ready State can have any no. of process

	min	max	max
Ready	0	any	any
Run	0	1	P
Block	0	n	n

→ if P CPU's then

*) Process scheduling:
Process → Ready queue
Process → Device queue

*) Scheduler:
→ LTS (Hard disk → MM) (controls degree of multiprogramming)
→ STS (MM → ready)
→ MTS (Swap in / swap out) (decreases degree of multiprogramming)
To Ready Queue From CPU

*) Dispatcher:
Give CPU control to process selected by STS

Note: LTS selects good mix of CPU bound & I/O bound processes

(2)

*) Context switch:
(CPU switch among processes)
→ Save P_0 process context to PCB-P_0
→ Load state of P_1 process from it's PCB-P_1
→ Start executing P_1

*) Degree of multipro. = no. of process in M/M

*) Throughput = no. of process completed per unit time

#) Scheduling:

*) Terminologies:
→ AT: Process enter ready queue
→ BT: CPU time
→ CT: Finishes execn
→ TAT: CT-AT (or WT+BT = first time sys. time)
→ WT: TAT - BT
→ Response Time: First response - AT

*) Desired:
→ Maximize = CPU utilization, throughput
→ Minimize = TAT, WT, Response time

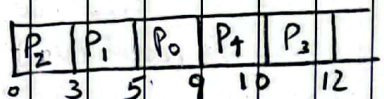
*) Non-Pre-emptive =
⇒ Completed execution
⇒ Need to perform I/O

Pre-emptive =
⇒ High priority process arrived
⇒ Time quantum expired

(Process leaves CPU when →)

#) Scheduling examples:

FCFS	AT	BT	CT	TAT CT-AT	WT TAT-BT
→ P_0	2	7	9	7	3
→ P_1	1	2	5	4	2
→ P_2	0	3	3	3	0
→ P_3	4	2	12	8	6
→ P_4	3	1	10	7	6
				5.8	3.4



#) FCFS, SJF/SRTF, Priority, Round robin, LJF, LRTF, HRRN, MLQ, MLFQ

*) FCFS: criteria = AT
mode = Non-Pre
Tie broken by: Process id
(Suffers from convoy effect)
Avg. WT ↑ ⇒ so execute smaller process before larger

*) SJF: (N.P) (SRTF is optimal)
SRTF: (Pre)
(Non implementable)
(Longer process suffers starvation)
criteria = BT
Tie broken by: FCFS

*) Priority: (NP & Pre)
(Starvation of low priority process is resolved by → Aging = Example: Priority increase by 1 after 2 clocks)
criteria = Highest priority
Tie broken = FCFC

*) Round Robin: (FCFS with preemption)
(No starvation)
criteria = TQ / AT (Provides best Avg. response time)
(Ready Queue circular)
mode = Pre

*) Longest Job first (Non-pre.)
LRTF (Pre.)
criteria = BT
Tie broken = Process id

3

3) *) Highest Response ratio next (HRRN)

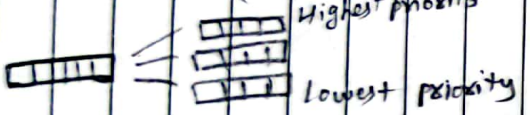
Criteria: Response ratio

mode = Non-pre.

$$\text{Response ratio} = \frac{WT + BT}{BT}$$

(Favors Shorter Jobs) (Limits WT of longer jobs)

*) Multilevel Q:



*) ML Feedback Q:

~~Refer Notebook for Scheduling~~

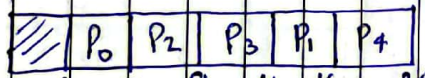
Note:) For minimizing response time, run jobs in increasing order of BT (or) Round Robin should be used

Note:) For minimizing TAT, run smaller jobs first.

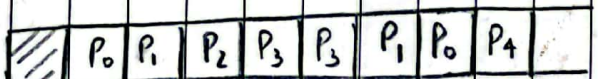
•) SJF & SRTF

	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
→ P_0	1	~~7~~ 6	8	16	7	15	0	8
→ P_1	2	~~5~~ 4	16	10	14	8	9	3
→ P_2	3	1	9	4	6	1	5	0
→ P_3	4	~~2~~ 1	11	6	7	2	5	0
→ P_4	5	8	24	24	19	19	11	11
					10.6	9	6	4.4

SJF



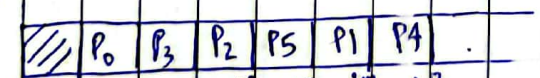
SRTF



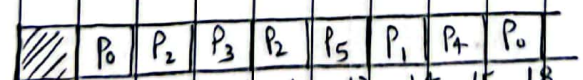
•) Priority

	AT	BT	Pr.	CT	TAT	WT
→ P_0	1	~~4~~ 3	4	5	18	4	17	0
→ P_1	(2)	2	(5)	17	14	15	12	13
→ P_2	2	~~3~~ 2	7	13	10	11	8	8
→ P_3	3	5	8(H)	10	8	7	5	2
→ P_4	(3)	1	(5)	18	15	15	12	14
→ P_5	4	2	6	15	12	11	8	9

non-pre.

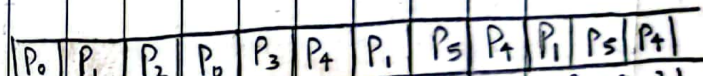


Pre.



•) Round Robin: TQ = 2 unit

	AT	BT	CT	TAT	WT
→ P_0	0	~~4~~ 2 ~~1~~	8	8	4
→ P_1	1	~~5~~ ~~3~~ 1	18	17	12
→ P_2	2	2	6	4	2
→ P_3	3	1	9	6	5
→ P_4	4	~~6~~ ~~4~~ 2	21	17	11
→ P_5	6	~~3~~ 1	19	13	10



~~P0 P1 P2 P0 P3 P4 P1 P5 P4 P1 P5~~ P4

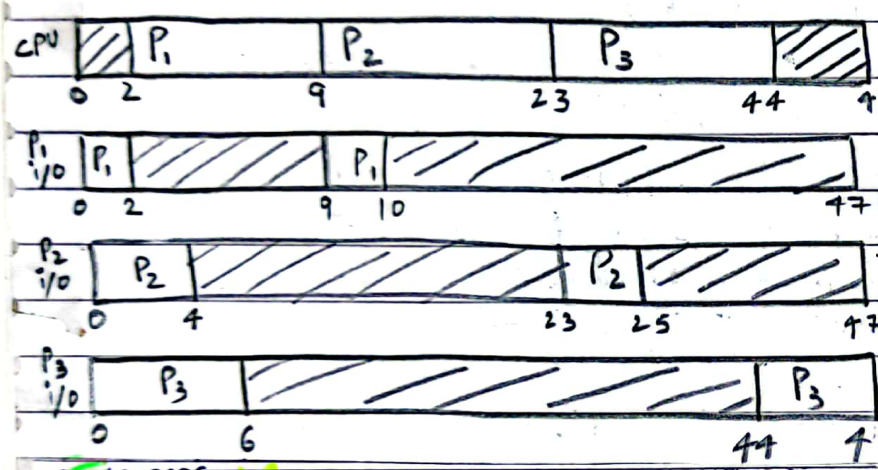
Q:) SRTF

AT		CPU	I/O	CPU
→0	P_1	2	7	1
→0	P_2	4	14	2
→0	P_3	6	21	3

CPU idle % $= \frac{(2+3)}{47} \times 100$

$= 10.63$

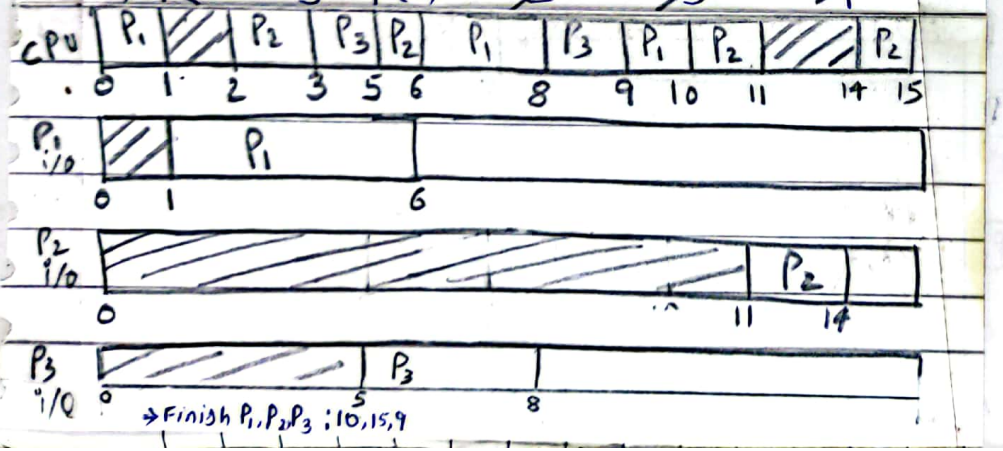
✓ B) 10.6



Gate 2006

Q:) Pre-Priority

	AT	Priority	CPU	I/O	CPU
→ P_1	0	2	1	5	1
→ P_2	2	3 (L)	2	3	1
→ P_3	3	1 (H)	2	3	1



→ Finish P_1, P_2, P_3 : 10, 15, 9

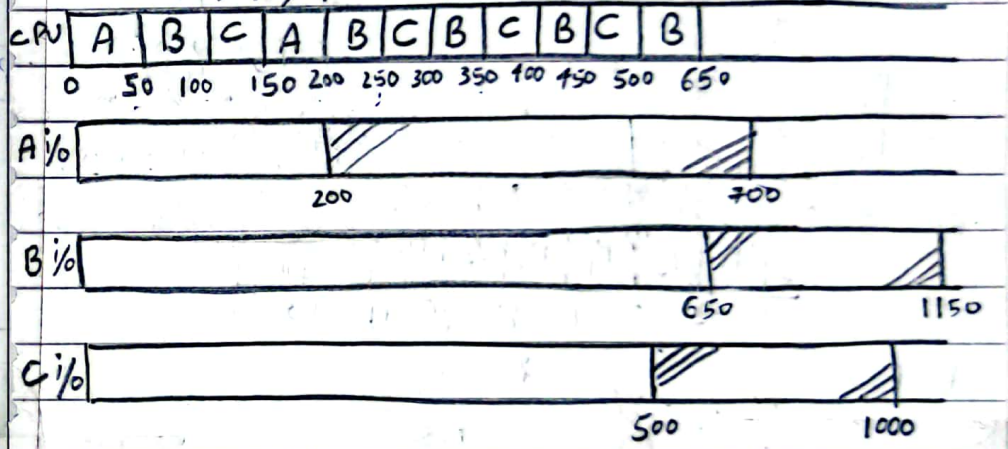
Gate 2014

Q:)

AT	P.no.	CPU	I/O
→0	A	100 50	500
→5	B	350 300 250 200 100	500
→10	C	200 150 100 50	500

T.Q = 50

A B C A B C B C B C B



C would complete it's I/O at 1000

Race cond.ⁿ: when o/p of process depends on execution sequence of process.

Process Sync.ⁿ:

*) General structure of process:

```
Pi()
{ while(T)
  { Initial Section
    Entry   "
    Critical "
    Exit     "
    Remainder "
  }
}
```

*) Criteria to Solve CSP:

1) Mutual exclusion = No two process present inside CS at same time.

2) Progress = जो process CS में जाना चाहता है उसे ही भेजो

3) Bounded wait: इतने से ज्यादा no. नहीं होने चाहिए

*) Solutions to CSP:

1) Two process solution
   → Boolean variable turn
   → Boolean Array flag
   → Peterson's solution

2) OS solution:
   → counting semaphores
   → Binary "

3) H/w soln. → Test & set lock

→ Boolean variable turn:

turn = 0/1

P_0	P_1
while(1)	while(1)
{while(turn≠0);	while(turn≠1);
CS	CS
turn = 1;	turn = 0;
Remainder	Remainder
}	}

- जब तक मेरा turn नहीं आएगा, I'll busy wait
- मेरा CS होगया, I'll give turn to someone else
- ME ✓
- Prog. ✗ (Bec. Strict Alternation)
- Invalid soln.

→ Boolean Array flag:

Flag = F F

P_0	P_1
while(1)	while(1)
{flag[0] = T	{flag[1] = T
while(flag[1]);	while(flag[0]);
CS	CS
flag[0] = F;	flag[1] = F
Remainder	Remainder
}	}

- I want to go in CS
- I'll wait till my brother is in CS
- मेरा CS होगया, I'm not interested to go in CS

- ME ✓
- Prog ✗ (Bec. Deadlock occurred at CS = 0)
- Invalid soln.

→ Peterson's solution:

turn = 0/1 Flag = F F

P_0	P_1
while(1)	while(1)
{	{
flag[0] = T	flag[1] = T
turn = 1	turn = 0
while(turn==1	while(turn==0
&& flag[1]==T);	&& flag[0]==T);
CS	CS
flag[0] = F	flag[1] = F
Remainder	Remainder
}	}

- I'm interested to go in
- I'll give turn to my brother
- Brother's turn && He wants to go in? (I'll busy wait if YES && YES)
- my CS done, I'm no longer interested
- ME ✓
- Prog. ✓
- Bounded wait ✓

*) Semaphores: (Binary)

wait(S) or P(S)

```
{while(S<=0);
 S--;
}
```

signal(S) or V(S)

```
{ S++;
}
```

5

```
Pi()
{ while(T)
  { Initial section
    wait(S)
    CS
    signal(S)
    Remainder
  }
}
```

Semaphores Ensure ME ✓ Prog ✓ but not Bounded wait

*) Ordering using Semaphores:

$P_2 \rightarrow P_3 \rightarrow P_1$ $S_1=0$ $S_2=0$

$P_1()$	$P_2()$	$P_3()$
$P(S_2)$	↓	$P(S_1)$
code	code	code
	$V(S_1)$	$V(S_2)$

*) Deadlock: To avoid deadlock, order of wait operation on different semaphores among different process may be same i.e.

P_1	P_2
wait(S_1)	wait(S_1)
wait(S_2)	wait(S_2)

*) Classical Sync.ⁿ problems-

1) Producer Consumer

S=1 E=n F=0

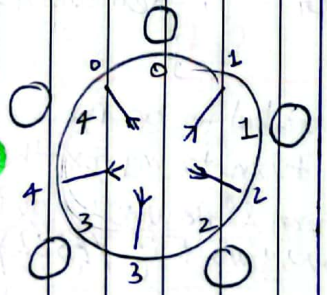
```
Producer()
{while(T)
 {//Produce an item
  wait(E)//overflow
  wait(S)//CS
  //Add item to buffer
  signal(S)//CS
  signal(F)//count++
 }
}
```

```
consumer()
{while(T)
 { wait(F)//underflow
   wait(S) //CS
   //Pick item from buffer
   signal(S)//CS
   Signal(E)//count--
   Consume item
 }
}
```

2) Dining Philosophers:

```
Philosopher()
{while(T)
 { Thinking();
   wait(chopstick[i]);
   wait(chopstick[(i+1)%5]);
   Eat();
   Signal(chopstick[i]);
   signal(chopstick[(i+1)%5]);
 }
}
```

wait(x) signal(x)



Invalid sol.ⁿ due to posibility of dead lock (सब सारे अपना Right बाया उठा लें)

Proposed solution:

1) Allow Atmost 4 philosophers at a time.
2) Allow 6 chopsticks
3) pick only if both chopsticks are available (i.e using semaphores)
4) Any one philosopher picks up right chopstick first & then Left chopstick. (ie ~~First Eth~~ order change करें)
5) odd philosopher → Left chopstic then right ,,
   Even ,, → Right ,, then Left ,,

3) Reader-writer:

mutex=1 wrt=1 } Semaphores → Sync. b/w RR; Sync. b/w WW, WR, RW

readCount=0 > variable to count no. of Readers

```
writer()
wait(wrt)
CS//write
signal(wrt)
```

```
Reader()
wait(mutex)
readCount++
if(readCount==1)
   {wait(wrt)}
signal(mutex)
CS//Read
wait(mutex)
readCount--
if(readCount==0)
   {Signal(wrt)}
signal(mutex)
```

*) Counting semaphores:

Binary semaphores suffer from busy wait, so, counting semaphores were introduced.

```
Struct{
   int value=0;
   Struct process *list;
   }semaphore;
wait(semaphore *S)
{ S->value--;
  if(S->value<0)
   { Add this process to S->list;
     block(); //Suspend
   }
}
signal(semaphore *S)
{ S->value++;
  if(S->value<=0)
   { Remove a process from S->list;
     wakeup(P);
   }
}
```

*) H/w sol.ⁿ) Test & set lock
Disable interrupt

H/W Solution :-

- Test & set lock
- Disable interrupt

M1:) Binary variable Lock

Lock [0] 0 → vacant cs, 1 → full cs

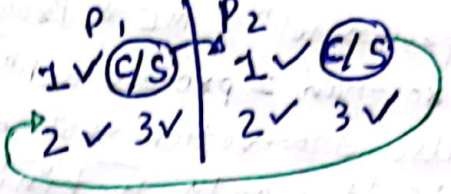
```
1. while(Lock == 1);
2. Lock = 1
```
Entry code

3. CS

```
4. Lock = 0
```
Exit code

- ME ✗ Deadlock free ✗

Reason :-

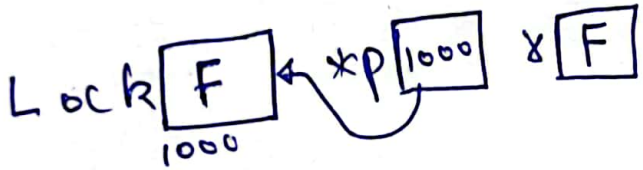


- It is a multi process sol.ⁿ

M2:) Test & Set lock (makes entry code atomic)

```
entry code [while( test_and_set(&lock));
           CS
exit code  [Lock = F
```

```
bool test-and-Set(bool *P)
{ bool r = *p;
  *p = T
  ret.(r)
}
```



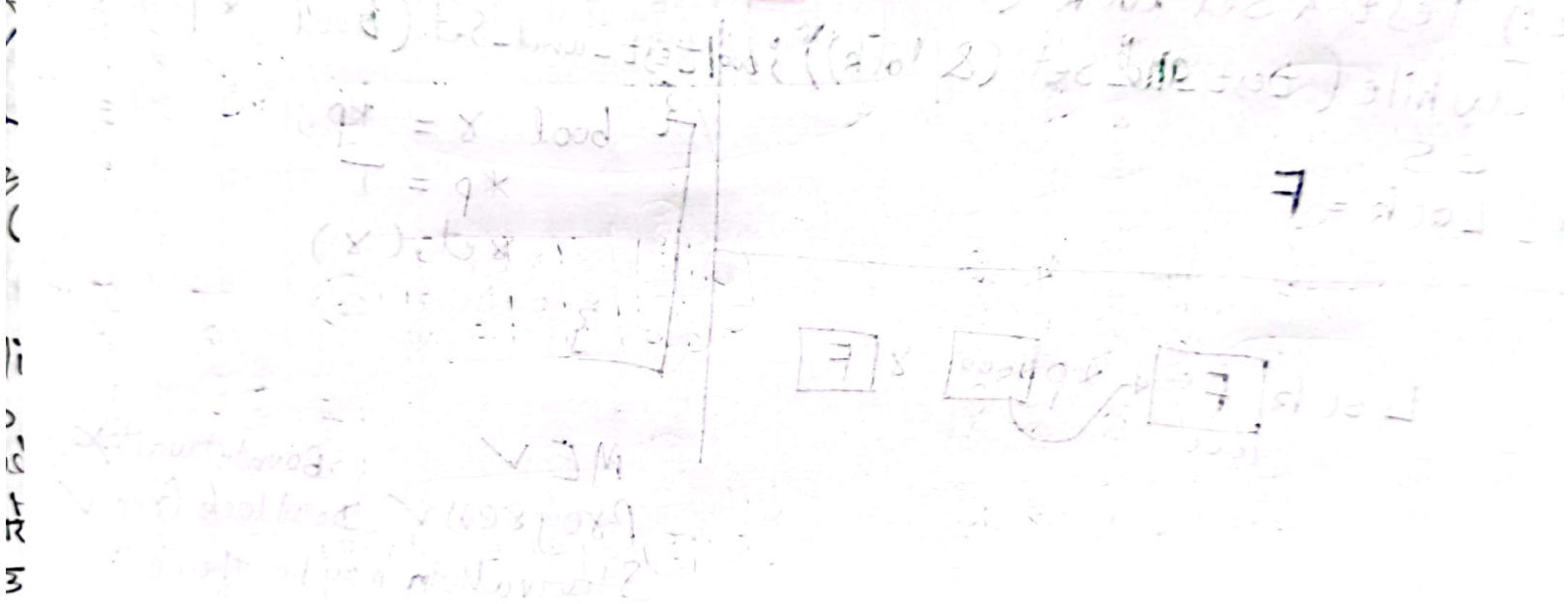
ME ✓ Bounded wait ✗

Progress ✓ Deadlock free ✓

Starvation may be there

Note:
- All semaphore operations are atomic, implemented in Kernel.
- Interrupt priority:-
  CPU > HDD > mouse > keyboard

Note:
- For 2 process soln: ME & Progress are mandatory criterias.
- For more than 2 process soln: All 3 mandatory criterias.
- It is also starvation that a process is not being able to finish, even if it wants to finish.
- It is also starvation that you're not being able to enter CS & someone else is taking multiple rounds continuously.



*) Deadlock:

*) Necessary conditions:

1) Mutual exclusion = Resource is one, but desired by more than 1 process, but allocated to 1 process at a time.

2) Hold & wait = Process holding one resource & requesting for another resource held by another process.

3) Non-preemption = Resource can't be preempted

4) Circular wait = Processes waiting for resource in circular manner

$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$

*) Handling methods:

1) Prevention: Design protocols to guarantee no possibility of deadlock

Ex: (B. Algo)

2) Avoidance: Avoid deadlock in runtime

3) Detection: Detecting when deadlock has occured

4) Ignorance: Pretend deadlock never occurs

I) Prevention:

a) Mutual Exclusion: No solution (bcz. h/w property)

b) Hold & wait: Sol 1) Acquire all resources before running.

Sol 2) (जितने resource चाहिए उतने तो पहले से ही हो, additional और resource चाहिए तो पहले release करो सारे को, (market में दिखाओ)

Sol 3) wait timeout

c) Non-preemption = एक process से resource prempt करा सकते है जो Run नहीं कर रहा And waiting for some other resources.

d) Circular wait = Process could either request resources in increasing order or decreasing. (Head on collision)

II) Avoidance (Banker's Algo):

	Max need			Allocation			current need		
	E	F	G	E	F	G	E	F	G
P_0	4	3	1	1	0	1	3	3	0
P_1	2	1	4	1	1	2	1	0	2
P_2	1	3	3	1	0	3	0	3	0
P_3	5	4	1	2	0	0	3	4	1

System max			Available		
E	F	G	E	F	G
8	4	6	3	3	0

3	3	0
1	0	1
4	3	1
1	0	3
5	3	4
1	1	2
6	4	6

Safe sequence = $P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$

III) Detection:

Active approach / Lazy approach

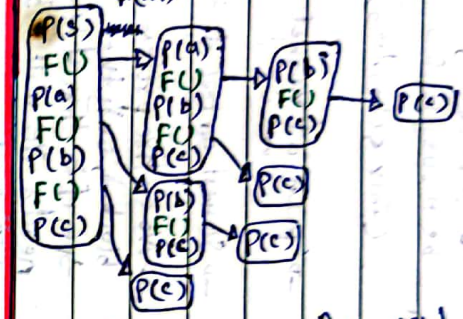
(Detection Algo frequently) (Detection Algo when deadlock)

Recovery = Process Termination
Prempt resources

IV) Ignorance = Pretend deadlock will never occur

#) Fork:

Ex:)



o/p: Sabccbccabccbcc

7

- no. of fork() => $2^n - 1$ child process
  2^n Total process

Note:

If, no. of process = 3
no. of resources = 4
max. m demand of a process = K
To avoid deadlock, $K_{max.} = ?$

$4 \geq 3(K-1) + 1$

$2 \geq K \Rightarrow K_{max.} = 2$

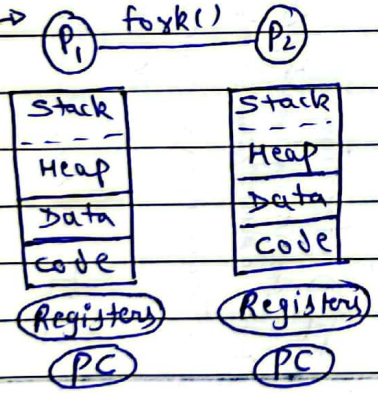
*) Process / Threads / user mode / Kernel mode:

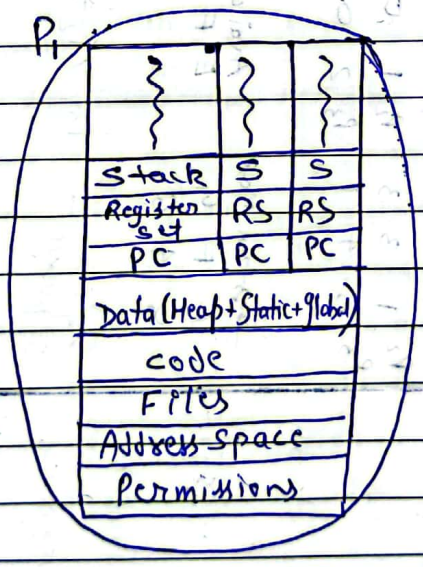
- System call:
  - → Trying to call a fx" which is present in OS.
  - → c/s needed
  - → If something is controlled by OS, we can ask it by system call.
- Foxk:
  - → A system call used to handle many requests at same time.
  - → c/s needed.
- Thread:
  - → One process in which many tasks are there
  - → data (Static + global) } common
    Heap
    code
  - → Stack
    register set
    PC } it's own
  - → c/s not needed

- Process vs Thread (ULT)
  - → System calls involved → No system call
  - → c/s ✓ → c/s ✗ (Register set switch ✓)
  - → ~~Different~~ copies of code & data → Same copies of code & data

- ULT vs KLT
  - ULT → Created by app^n packages/libraries w/o taking any support from OS (unfair scheduling)
  - KLT → Scheduling must be through OS (fair scheduling)
  - ULT → Thread mngment is done at user level
  - KLT → Run at kernel mode
  - ULT → Fastest context switch
  - KLT → Slowest context switch
  - ULT → Entire process gets blocked when a thread require an i/o device or any system call
  - KLT → No entire process is blocked, only the specific thread which needs i/o will be blocked.

ULT are visible to kernel
Kernel is unaware of ULTs
Only 1 ULT is in running state at a time

→ 



Note:) About foxk() :-

- It returns 0 to the child process & returns 1 to the parent process

Ex:)
```
if (foxk() == 0)
  ...
```

```
if (1 == 0)
  ...
```
→
```
if (0 == 0)
  ...
```

- If after foxk(), no code is there, an empty child process created.

Ex:)
```
foxk();
//no code
```
→ //Empty ⇒ 1 child process

- Parent process can execute parallely with child process or maybe after execution of child process
- Child process can be duplicate of parent or maybe loaded with other program.

Memory Management:

Addressing:

Ex: N = #cells/words in MM = 16
n-bit address = $\lg_2 N = 4$

Ex: RAM = 500 GB; 1 word = 1 B (i.e. 8 bit processor)
$N = 500 \times 2^{30}$ cells/words
n = 39

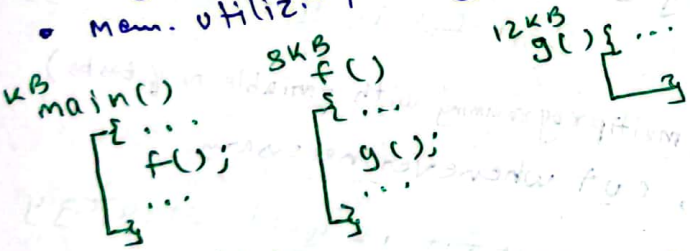
Ex: N = 128 GW
m = 32 b = 4 B

RAM capacity	Addressability
word: 128 G	n = 37 bits
Byte: 128×4 B	n = 39 bits
bits: 128×4×8 b	n = 42 bits

Program linking & loading:-

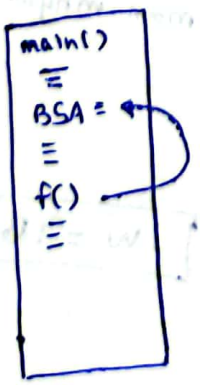
→ Static loading:-
- All modules of code are loaded into MM before runtime. (30 KB)
- mem. utiliz.n ↓ Execution speed ↑
- Multiprocessing ↓

→ Dynamic loading:-
- Only load modules of code that are currently needed. (10 KB)
- mem. utiliz.n ↑ Exec.n speed ↓



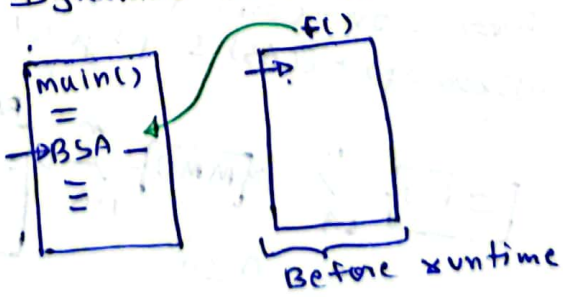
→ ~~Dynamic~~ Linking:-
- Binding address of called fx.n in calling fx.n's MLL code.
- main
f()
Branch and Save Address

- Static linking



Static loading

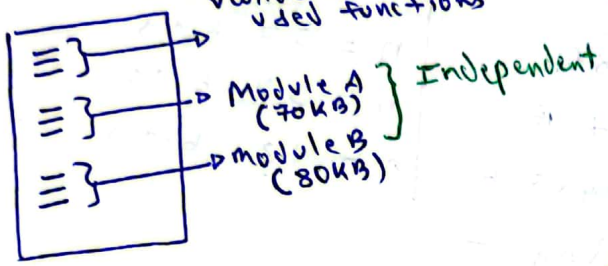
Dynamic linking



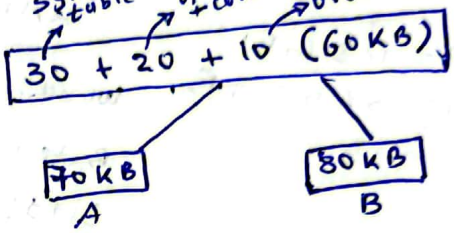
*) Memory management :-

contigyous	Non-contigyous
1. Overlays	1. Paging
2. Partitioning	2. Segmentation
3. Buddy System	3. Segmented paging
	4. Virtual mem. & demand paging

*) Overlays: Global variables + commonly used functions



overlays tree :-



max^m mem. req. = 60KB + 80KB = 140KB
min^m mem. req. = 60KB + 70KB = 130KB

*) Partitioning:

a) Fixed partitioning (MFT: multiprogramming with fixed no. of tasks)

- # of partitions & size of each partitions is fixed.
- max^m no. of processes (at any time) = # partitions
- Protection: via limit registers
- internal frag. ✓
  External frag. ✓

Best fit is good strategy

b) variable partitioning (MVT: multiprogramming with variable no. of tasks)

- One large chunk available, cut whenever necessary
- ~~internal~~ internal frag. ✗
  external frag. ✓

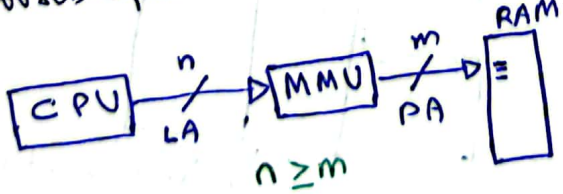
worst fit is good strategy

•) Compaction: One of the solution to external frag., here we shuffle the user memory such that all free space is aggregated at one chunk.

*) Basics for non-contigyous memory allocation:

•) Logical Address Space (LAS) & Physical Address Space (PAS) :-

MMU = mem. mngmnt unit



$n \geq m$

If LAS = 2^{20} words or B ⇒ LA = 20 bit
If PAS = 2^{16} words or B ⇒ PA = 16 bit

1W = 1B

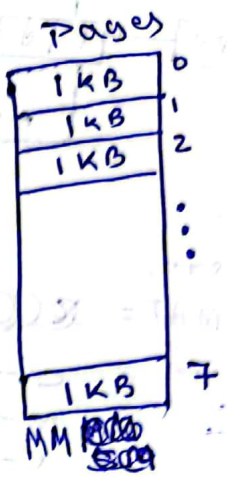
*) Paging : MM divided into pages of equal size

Page size = PS = 1KB

Ex:) LAS = 8KB ⇒ LA = 13b

PAS = 4KB ⇒ PA = 12b

i) Organization of LAS: Division into pages

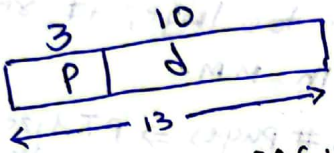


#pages = $N = 2^p$; p = bits for page no.

$N = \frac{LAS}{PS} = \frac{8KB}{1KB} = 8$ (i.e. 0 to 7)

$\Rightarrow p = 3$

⇒LA:



p = page no.

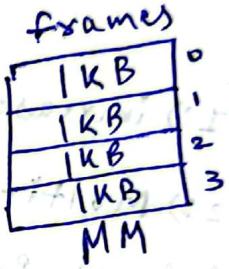
d = page offset

ii) Organization of PAS: Division into frames

Frame size = Page size

⇒ FS = 1KB

#Frames = $M = \frac{4KB}{1KB} = 4$ (i.e. 0 to 3)



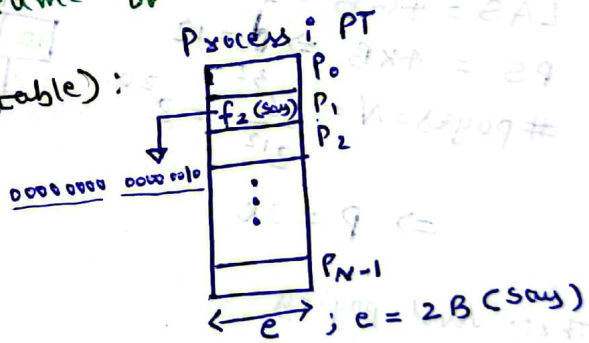
⇒ PA:

f = frame no.

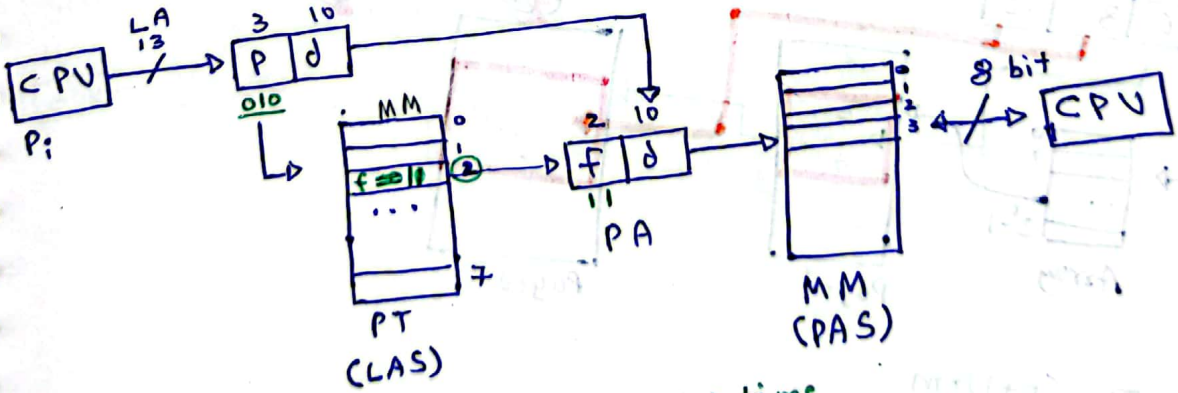
d = frame offset

iii) Organization of MMU (Page table):

- Every process has a PT.
- PT size = N x e
  = 2^p x f bits



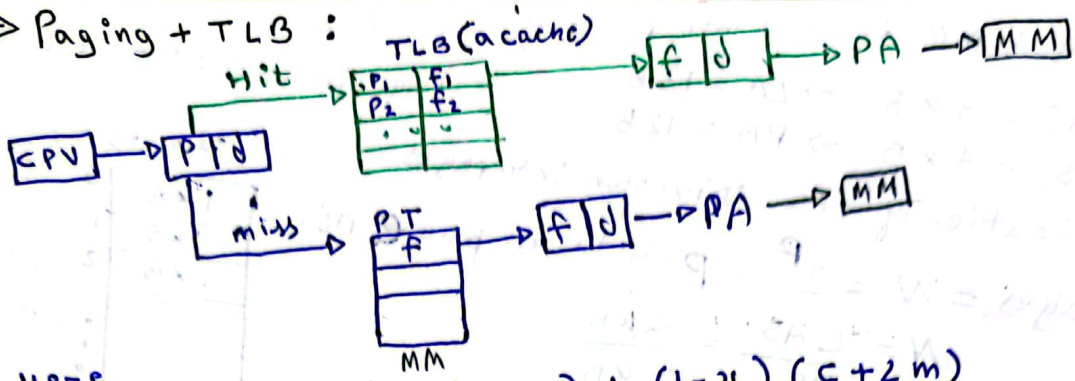
iv) Translation :



⇒ EMAT = 2m ; m = mem. access time

To reduce this EMAT we use Paging + TLB

→ Paging + TLB :



Here,

EMAT = $x(c+m) + (1-x)(c+2m)$

c → Time to search in TLB (cache access time)

m → Time to search in MM (MM access time)

x → TLB hit ratio (0 to 1)

→ Problem in simple paging: If PT size too large, it requires huge space in MM.

Sol-1:) increase page size ⇒ Reduction in # pages ⇒ PT size reduce ⇓ Internal frag.

Sol-2:) Multi-level paging:

→ Multi-level Paging :

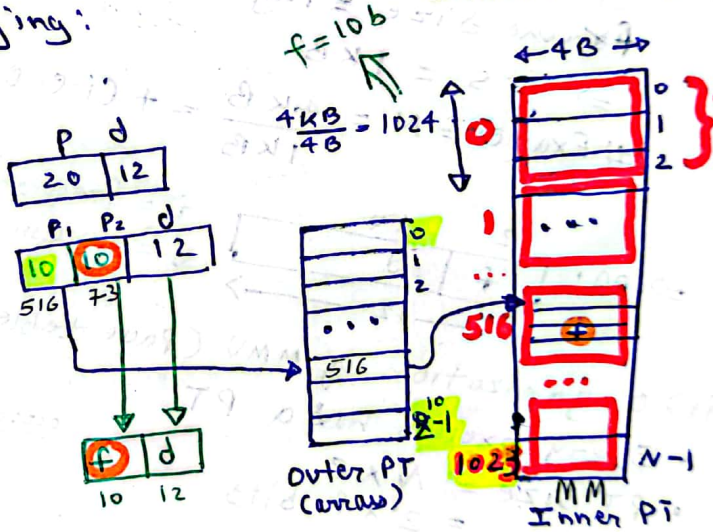
•) Ex: Two level paging

LAS = 4GB ⇒ LA = 32

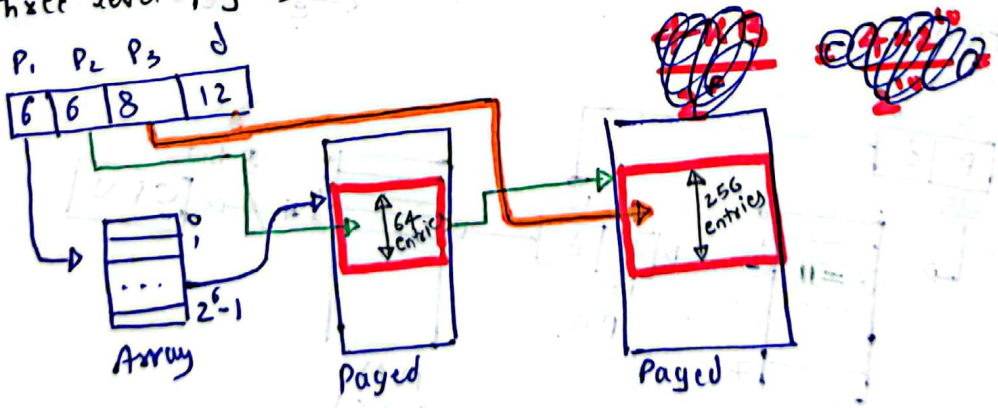
PS = 4KB ⇒ d = 12

$\#pages = N = \frac{2^{32}}{2^{12}} = 2^{20}$

⇒ P = 20



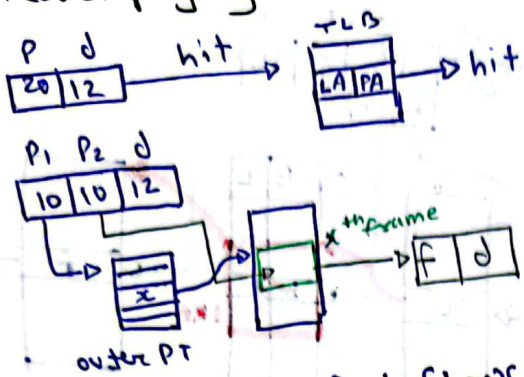
•) Ex: Three level paging



•) EMAT = (n+1)*m
(n-level paging)

→ Multilevel paging with TLB

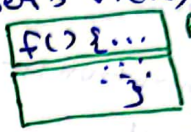
•) Ex:



•) EMAT = x (c+m) + (1-x) (c + [n+1] m)

(n-level paging + TLB)

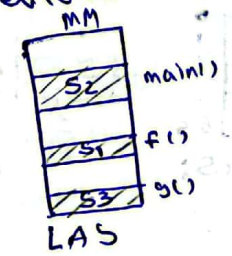
*) Segmentation: • User's view (i.e, programmer's view) is not preserved with paging. So, segmentation came.



• MM devided into unequal sized segments

• Organization of LAS:-

→ LAS is divided into unequal dized segments.

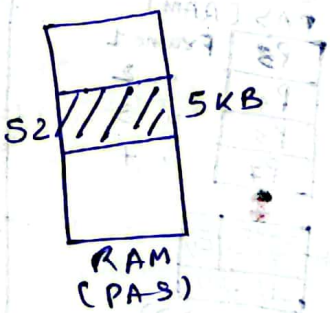


• ~~Regio~~ Organiz.ⁿ of PAS:-

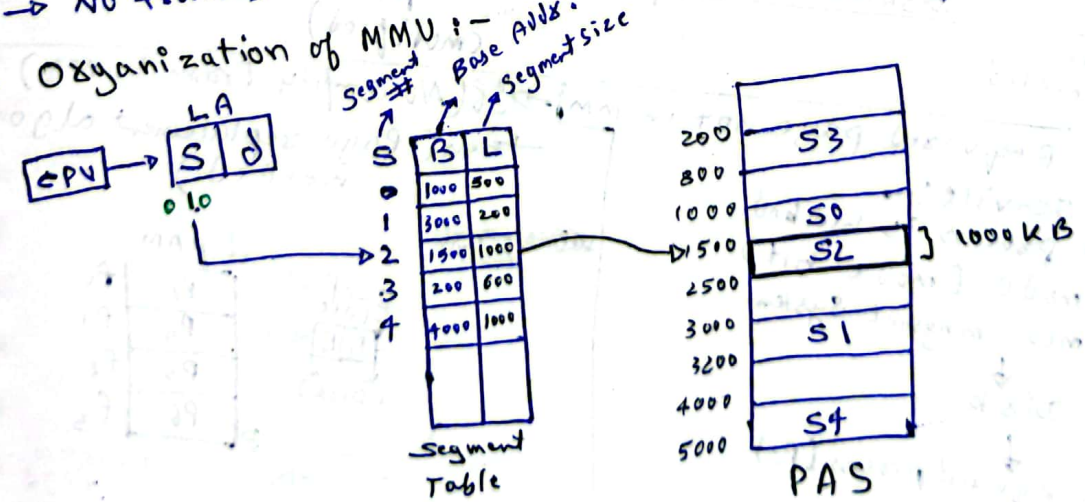
→ variable sized partitioning is applied on RAM

→ segment is loaded in a fashion of contigous memory allocation.

→ No frames



• Organization of MMU :-



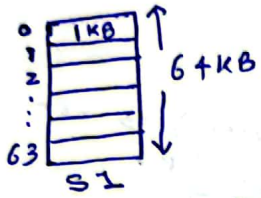
Segmented paging:

- Paging is done on segments
- Every segment have a PT
- Ex:)

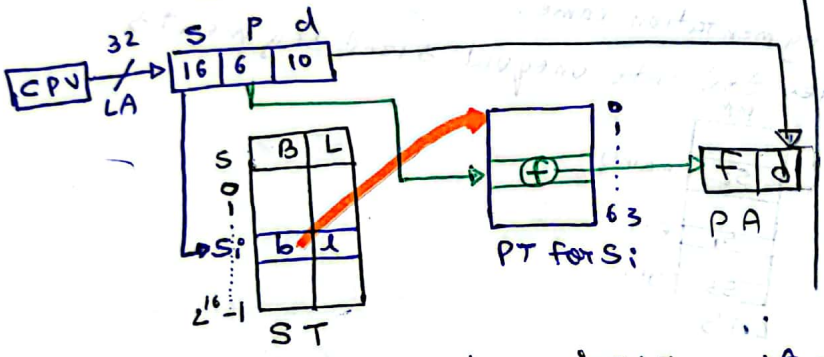
  Page Size (PS) = 1KB

  Segment S1 = 64KB

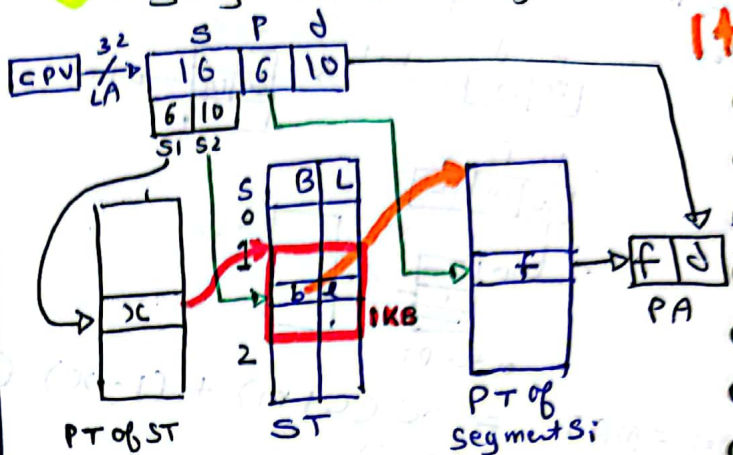
  # pages in segment S1 = $\frac{64KB}{1KB}$ = 64



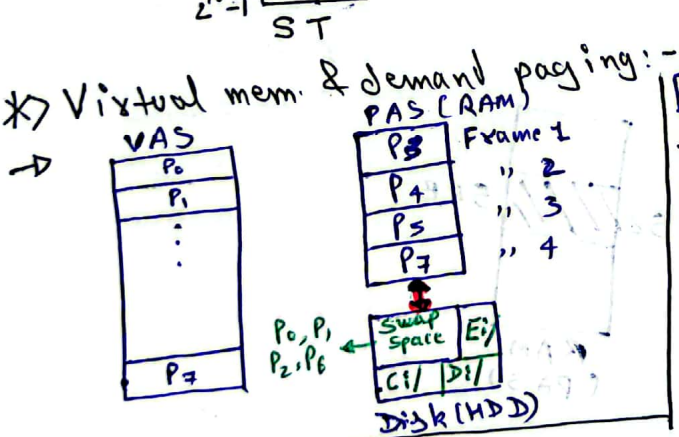
- MMU organization:-

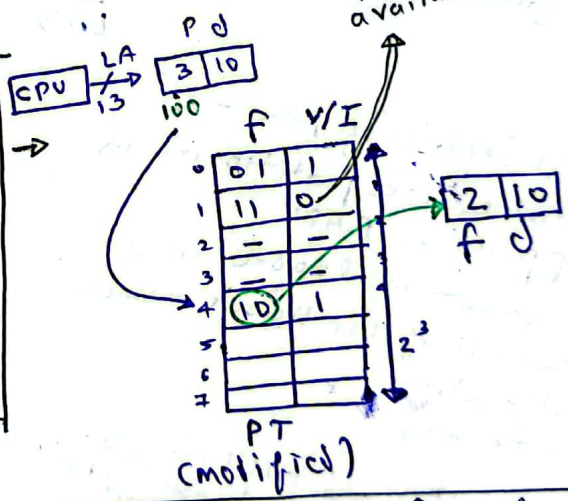


Paging the ST (segment table):



Virtual mem. & demand paging:-





F: Frame no.

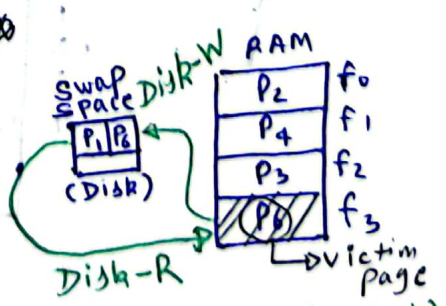
V/I: valid / Invalid

- Page fault: Required page not in MM
- Page fault service:-
  - Current process is blocked
  - Kernel mode (mode-bit)
  - virtual mem. mngmnt System

  ↓ Disk

  ↓ Disk gives required process (Pi) to RAM

- gf (Empty frame available in MM) then (just move process from swap space to RAM)
- gf (No empty frame in MM) then (Page replacement algo needed)



(Say P1 we want to load)

Dirty page ⇒ Disk-R, Disk-W

clean page ⇒ Disk-R, but avoid Disk-W cuz you can load P6 again from Loader

→ Pure demand paging:
(Default) No one in MM, initially.

→ Pre-fetch demand paging:
some pages are pre-loaded initially into PAS, before start of process.

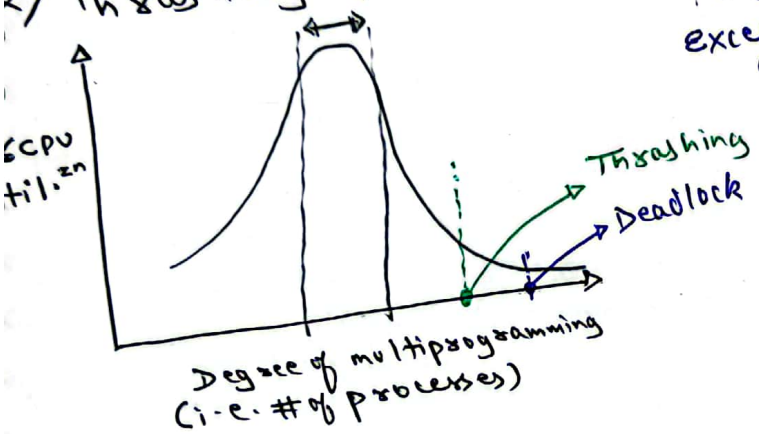
→ In virtual mem. + Demand paging:

$EMAT_{DP} = p\$ + (1-p)(2m)$;

MM access time = m
Page fault servile time = PFST = $

Page fault rate = p
Page hit rate = 1-p

◇ Thrashing: (MM spends lots of time handling page faults)



process ↑ ⇒ PAS ↓ & vice versa

Excessive page activity ⇒ v.high page fault rate

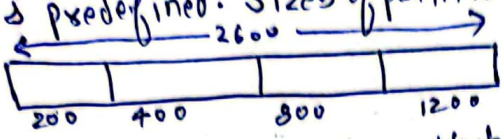
Reasons :-
1) Primary: Too few frames
Too many processes
2) Secondary: Page replacement algo.
Too small page size

Note:
Paging EMAT = 2m ; m = main mem. access time
Paging+TLB EMAT = h(c+m) + (1-h)(c+2m)
Multilevel Paging EMAT = (n+1) x m ; n = no. of levels
Multilevel Paging + TLB = h(c+m) + (1-h)(c + [n+1] x m)
Demand paging EMAT = p · PFST + (1-p)(2m) ; p = page fault rate

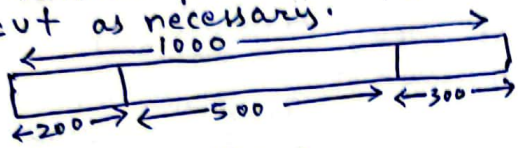
*) Managing main memory –

1) Contiguous:–

a) Fixed partition: No. & size of partitions is predefined. Sizes of partitions may differ.



b) variable partition: Large block available, cut as necessary.



•) Space allocation:

a) First fit = First which is capable enough
b) Best fit = Smallest " " " "
c) Worst fit = Largest " " " "
d) Next fit = After satisfying a request, in best fit, we start satisfying next request from current position.

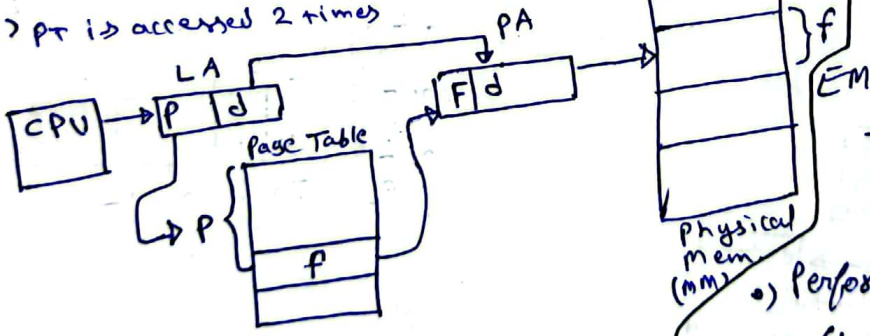
•) Solutions to external fragmentation:

1) Compaction = Shuffle memory contents so as to place all free memory together in one large block.
2) Non-contiguous Allocation (Paging)

2) Non-contiguous :–(Paging)

•) Secondary M devided into pages
MM " " Frames
Frame size = Page size

•) PT is accessed 2 times

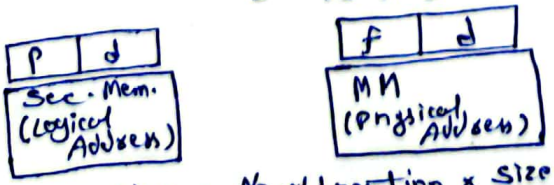


p = Page no.
d = instruction offset (Line no. inside page)
f = frame no.

•) Page size = 2^d ; No. of Pages (of Process) = $2^p = x$

Page table size = no. of entries in Page table × Size of each entry
= $2^p \times$ PTE (in bytes)

Process size = no. of pages × size of each page
= $x \times 2^d$



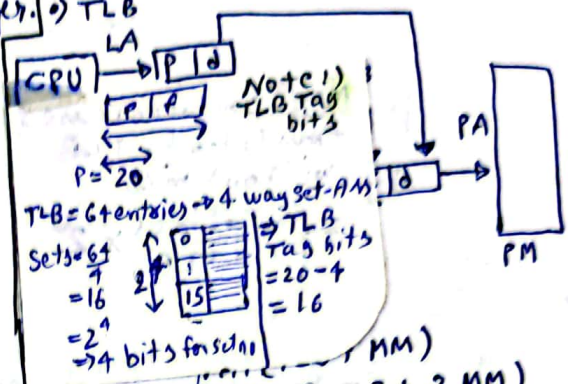
Memory size = No. of Location × size of each location

No. of Locations = $\frac{\text{mem. size}}{\text{Size of each location}}$

Ex:) <Byte addressable>

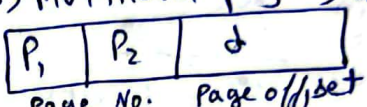
SM	LA	MM	PA	p	f	d	Page size
32GB	35	128MB	27	25	17	10	1KB

•) TLB

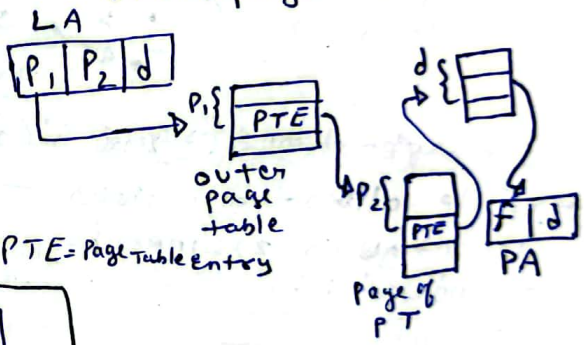


EMAT = Hit (TLB + MM)
+ (1 − Hit) (TLB + 2 MM)

•) Multilevel paging (helps to reduce size of PT)



P_1 → index into outer page table
P_2 → displacement within page of outer page table.



EMAT = hit (TLB + MM)
+ (1 − hit) (TLB + 3 MM)

•) Performance of demand paging:

EMAT = (1 − p) × m_a + p × page fault service time

p → Page fault rate / probability / hit ratio
m_a → memory access time

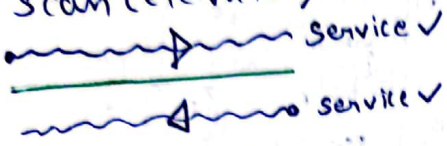
•) Page replacement –

1) Fifo (belady) (Page faults may increase as no. of frames increase)
2) Optimal = replace page not used for longest period of time in future.
3) LRU
4) LFU : Replace page with smallest count
MFU :

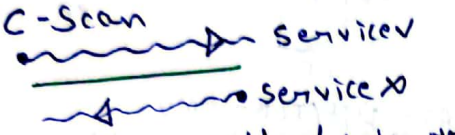
•) Thrashing: Process spending more time on paging than executing bec. of high degree of multiprogramming.

Disk Scheduling:

1) FCFS
2) SSTF
3) scan (elevator)

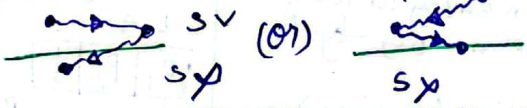


4) C-Scan



5) Look : inspite of going to the end, goes only to the last request to be serviced in front & then reverses the dir.n from there only.

6) C-look : inspite of going to the end goes only to last request to be serviced on both sides



Note: Threads; they :- share: code, data (static storage), files, Address space, Permissions
Have own: Registers, Stack, Program Counter

17

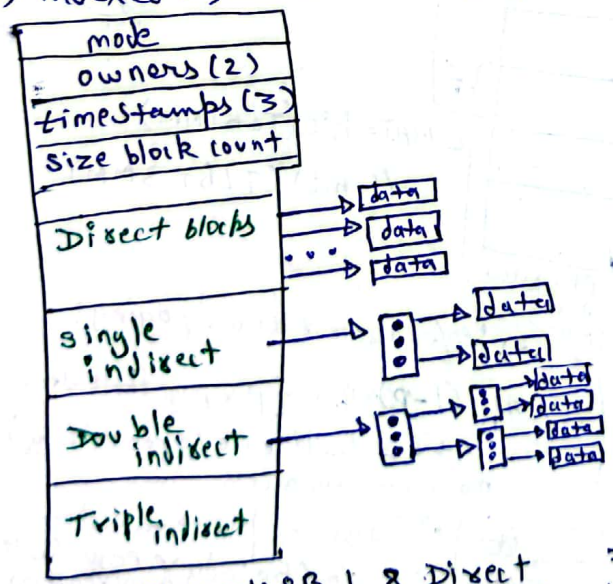
Note:

Note: About page replacement:

- compulsory page fault ↓ count ↑ (CTO find no. of page faults)
- Belady anomaly: increasing # page frames sometimes results in an increase in page faults.

File allocation methods :-

1) contiguous 2) Linked } no external frag.n
3) indexed : }



Ex:) Block: Size = 128 B
Address = 64 b = 8 B

8 Direct = 8 × 128 = 1 kB
1 Single indirect = 1 × 16 × 128 = 2 kB
1 Double " = 1 × 16 × 16 × 128 = 32 kB
⇒ File size = 35 kB

Address per block = $\frac{128\ B}{8\ B}$ = 16

#) Disks:

- First, revise it from <COA> P.no - 2 & 3
- TT (Transfer Time) = Time to read from the sector to MM
- TTT = Disk i/o time

- Q:) 16 Platters
  (2 surface each)
  2k Tracks per surface
  512 Sectors per track
  2KB Data per sector
  ST = 30 ms
  RPM = 3600

  Find a) Unformatted capacity
  b) I/O time for reading a sector
  c) Effective data transfer rate (in Bytes/s)

a) $16 \times 2 \times 2K \times 512 \times 2KB = 64 GB$

b) TTT = ST + RL + TT

RL:-
3600 r → 60 s
1 r → $\frac{1}{60}$ s
RL = $\frac{1}{120}$ s
= 8.3 ms

TT:-
1 r → 1 track data read
$\frac{1}{60}$ s → 512 × 2KB
$\frac{1}{60 \times 512}$ ← 2KB
= 0.0325 ms

TTT = 30 + 8.3 + 0.0325

c) 1 r → 512 × 2KB → $\frac{1}{60}$ s

512 × 2KB × 60 ← 1 s
= 60 MB
⇒ 60 MB/s

- Q:) Load a program of 64KB where page size = 2KB
  ST = 30 ms
  1 r = 20 ms
  32KB → Track size
  TTT = ?

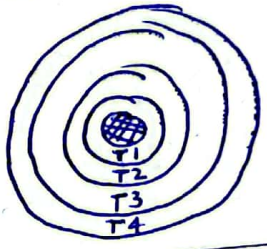
#pages = $\frac{64KB}{2KB}$ = 32

for each page:-
RL:- 10 ms
TT:- 1 r → 32KB → 20 ms
2KB → 1.25 ms

TTT = 30 + 10 + 1.25 = 41.25 ms

for 32 pages:-
TTT = 32 × 41.25 ms

Note:



Note: Constant Linear velocity (CLV)
Constant angular velocity (CAV)

CLV 

CAV 

From cylinder to cylinder
→ Density of bits = Uniform
→ # Sectors per track = increases
→ Used in modern CD/DVD

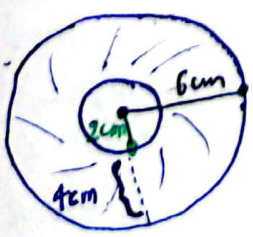
→ Density of bits: Dec.
→ # Sectors per track = Same for all tracks

Let, 1 Track = 10 MB

Capacity (of Surface): = 10 + 20 + 30 + ...
= 10 + 10 + 10 + ...
I st Track, II nd Track, III rd Track

- Q:) #Surfaces = 6
  inner diameter = 4 cm
  outer diameter = 12 cm
  } For each surface
  inner track space = 0.1 mm
  # Sectors per track = 20
  1 sector = 4KB
  ST = 20 ms
  RPM = 3600

  Find a) capacity of disk
  b) 1 Sector i/o time



Tracks = $\frac{\text{Disk width}}{\text{Track width}}$
= $\frac{4 cm}{0.1 mm}$
= 400

a) 6 × 400 × 20 × 4KB

b) TTT = ST + RL + TT

RL:-
3600 r → 60 s
1 r → $\frac{1}{60}$ s
RL = $\frac{1}{120}$ s
= 8.33 ms

TT:-
1 r → 1 track i/o
$\frac{1}{60}$ s → 20 × 4KB
$\frac{1}{60 \times 20}$ s ← 4KB
= 0.83 ms

Note: working set algorithm (Dynamically implemented page replacement policy)

Seq.: 1 2 3 1 2 4 1 4 2 1 5 1 2 4 2 1

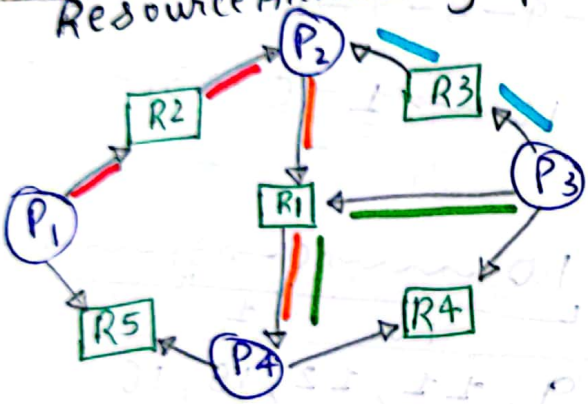
A = 4 (says)

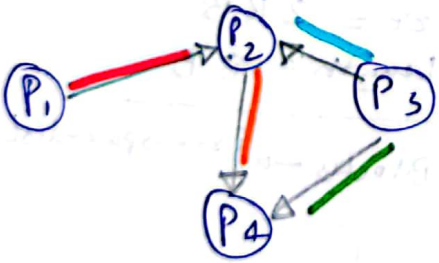
A → max^m window size allowed
w → my current window size

w = {1}
w = {1,2}
w = {1,2,3}
w = {1,2,3}
w = {1,2,3}
w = {1,2,3,4}
w = {1,2,4}
w = {1,2,4}
w = {1,2,4}
w = {1,2,4}
w = {1,2,4,5}
w = {1,2,5}
w = {1,2,5}
w = {1,2,4,5}
w = {1,2,4}
w = {1,2,4}

min^m window size = 3
max^m " " = 4

Resource Allocation graph to wait for graph :—





for PT 2)

total size = $2^{P3} \times 4B$
$= 2^9 \times 4B$
$= 2^{11} B$

$\Rightarrow$ Locations = $\frac{2^{36}}{2^{11}}$
$= 2^{25}$
$\Rightarrow y = 25$

for PT 1) Total size = $2^{P2} \times 4B$
$= 2^9 \times 4B$
$= 2^{11} B$

$\Rightarrow$ Locations = $\frac{2^{36}}{2^{11}}$
$= 2^{25}$
$\Rightarrow z = 25$