<OS>

→ acts as an interface b/n user & hardware of comp. system.

- functions of os:

convenient, efficient, reliable, Portable, scalable.

(Main goal)

- features of os:

Process, memory, device, file Management & protection and security

- Types of os:
  - uniprogrammed os { cpu idle ↑ throughput ↓
  - Multi prog... os { cpu idleness ↓ throughput ↑

preemptive | non-preemptive

preemptive	non-preemptive
A process can be forcefully taken out from CPU.	process can utilize CPU till it wants.

- Multi tasking / Time-sharing os

→ process will be executed on CPU in time sharing mode.

System calls:

- Interface b/n user & kernal.
- It is a f^n used to call os to avail some services.
- library f^n can call system calls to get services from os.

Comp. System Architecture (CSA)

Von-Neumon Archi- | Harvard Archi-

Process Management:

- Program: In context of os, it is (.exe file) created after compilation of program.

text.c → [compiler] → .exe (program)

- Process: Instance of program.
  - program in execution state.
    - doesn't mean running on CPU; it means utilizing CPU, memory, I/O.

Program	Process
① Passive entity	① Active entity
② doesn't use resources of comp. system.	② uses resources of comp. system
③ Reside in disk	③ Reside in MM (RAM).

★. Attributes of process:

Context of the process will be stored in PCB → Reside in MM.

Process id, process state, PC, LOF, LOD, protection inform^n.

★. States of process:

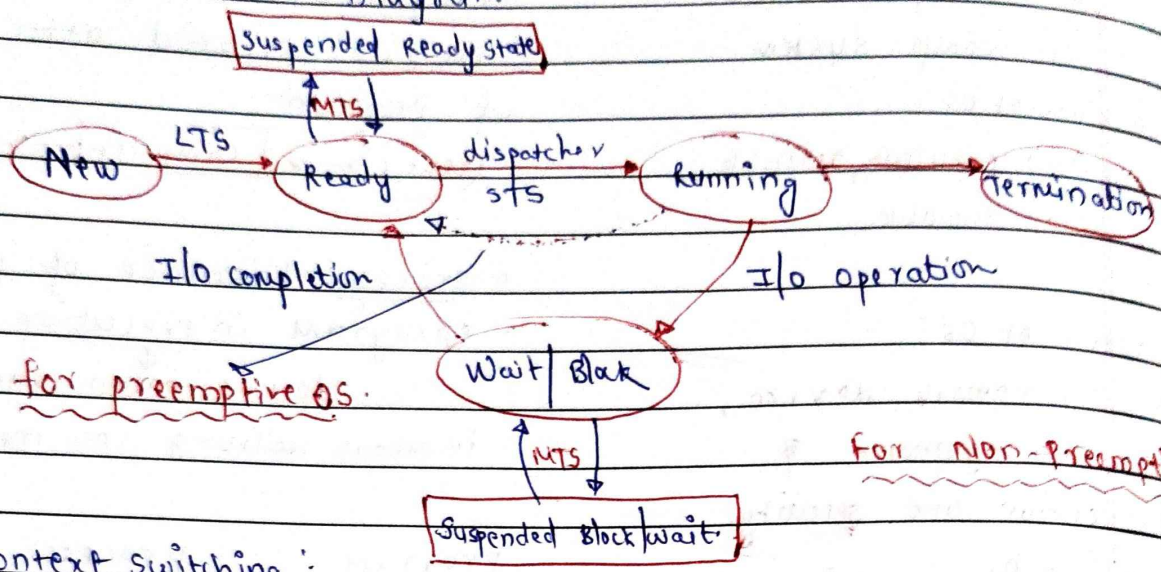
New, Ready, Running, waiting/block, terminate, suspended ready, Suspended block

- Degree of Multiprogramming:

→ No. of processes present in memory at a particular time

< preemptive ⊆ Non-Preemptive

★ State Transition Diagram:



- context switching:
  - → Each time when process moves from one state to another
    - → saving context of one process & loading context of another
  - → dispatcher is responsible for context switching

privileged (kernal mode)
- eg
  - set timer
  - context switching
  - clear memory
  - remove process from memory

Non-privileged (user mode)
- eg
  - Redding status of processor
  - Reading system time
  - sending final printout to printer

CPU scheduling:

→ process through which any one process can be selected from Ready state.

→ process have various times:

AT: time when process arrived to Ready state.

BT: time that process requires to execute.

CT: time when process completes its execution.

TAT: time when process is executed (TAT = CT - AT)

WT: total time process wait. (WT = TAT - BT)

RT: time diff. bt^n first response time & AT.

CPU utilization: % of time CPU is busy.

CPU throughput: Avg. no. of inst^n executed per unit time.

Types of CPU scheduling:

① FCFS:

criteria: AT ↓

mode: Non-preemptive

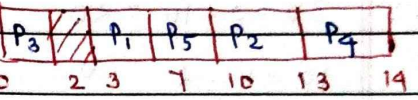
- No starvation.
- convey effect.
- In case of tie, process having smallest process id, executed first.

id	AT	BT
P_1	3	4
P_2	5	3
P_3	0	2
P_4	5	1
P_5	4	3

Ready queue:

time	process arrived
0	P_3
2	-
3	P_1
7	P_5 P_2 P_4

Running queue / Ganttchart:



② SJF:

criteria: BT ↓

mode: Non-preemptive

- starvation
- AWT ↓ (min)
- Max^m throughput ↑.

★ Exponential avg. formula used to predict the length of next CPU burst.

$$T_{n+1} = \alpha T_n + (1-\alpha)\, t_n$$

T_{n+1} → next predicted BT; T_n → current predicted BT; t_n → current actual time; α : 0 to 1

③ SRTF:

criteria: BT ↓

mode: Preemptive.

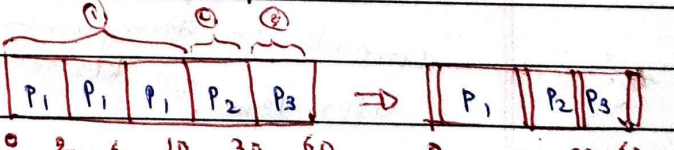
- starvation
- ATAT & AWT ↓ (min)
-

id	AT	BT
P_1	0	10
P_2	2	20
P_3	6	30

Ready queue:

time	process arrived
0	P_1
2	P_1 P_2
6	P_1 P_2 P_3

Running queue:



④ Round Robin:

criteria: Time quantum

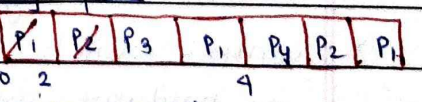
Mode: Preemptive.

- No starvation.
- RT ↑ (good)
- AWT & ATAT ↓ (poor)
- If time quantum of RR is greater than BT of all processes than it behaves like FCFS.

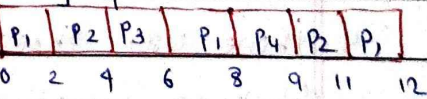
id	AT	BT
P_1	0	5
P_2	1	4
P_3	2	2
P_4	4	1

TQ = 2

Ready queue:



Running queue:



steps:

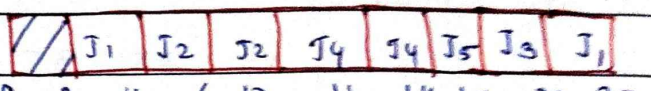
After adding process to gantt chart:

- Jis time pe add kiya uss time pe agar koi naya process aya ha to usko ready queue mo add kro pehla.

qn.

id	AT	BT	RT
J_1	2	10	0
J_2	4	6	0
J_3	6	5	11
J_4	10	4	0
J_5	11	3	3

SRTF



Throughput = No. of Jobs completed / total time taken.

= 5/30 = 166.67 Jobs/sec.

(WT): 30 - 2 = 10 = 20.

10 - 4 - 6 = 0

22 - 6 - 5 = 11

14 - 10 - 4 = 0

17 - 11 - 3 = 3.

Response time = First execution - AT time

WT = CT - AT - BT

#1 Threads: → user level, → kernal level

- component of process.
- lightweight process.
- A process may have multiple threads executing in it.
- It consists of: Thread id, Register set, stack, PC (don't share with other threads) → code section, Data section, etc. (share)

Adv:

- Responsiveness
- faster context switch
- effective utilization of multiprocessor system
- Resource sharing
- Communication.
- Enhanced throughput

~~user level threads~~ are transparent to kernal.

user level	kernal level
1. Implemented by user (easy)	1. Implemented by OS (difficult)
2. OS doesn't recognizes it	2. OS recognizes it.
3. context switch time ↓. & no hardware support reqd	3. context switch time ↑ & hardware support reqd
4. If one user thread performs blocking operation, then entire process will be blocked.	4. If 1 kernal is blocked, then other threads can still execute.
5. can't take advantage of multiprocessing	5. can take advantage of multiprocessing

Process Synchronization:

- It is a mechanism that deals with the synchronization of process.
- It controls the execution of process running concurrently to ensure consistent results are produced.
- used to avoid inconsistent results.

Problem w/o synchronization:

Inconsistency, data loss, deadlock.

Types of process

Independent | coordinating/cooperating.

Critical section:

- section of program where a process access the shared resources during execution.
- synchronization is only needed in C.S.

Race cond^n: undesirable sol^n that occurs when final result of concurrent processes depends on sequence in which they execute.

Synchronization mechanism:

- allows the process to access CS in synchronized manner.
- It adds: Entry section: gateway. ensures only one process must present at a time in CS. | Exit Section: exit gate.

process {
- Non C.S
- Entry section
- C.S
- Exit section
- Non C.S
}

sol^n to achieve the criteria:

① Software type sol^n:
- ① lock variable
- ② turn variable | strict alternation
- ③ Petersons sol^n.

② Hardware type sol^n:
- ① TSL
- ② Swap.

③ OS type sol^n:
- ↳ semaphore: Binary. Counting.

④ Compiler type sol^n:
- ↳ Monitors.

Criteria for synchronization mechanism

① Mutual Exclusion] Mandatory.
② Progress
③ Bounded wait] optional
④ Architectural Neutral

X → doesn't satisfies
✓ → satisfies.

Page No.
Date

① Software-type Soln:

① lock Variable:

```
Boolean lock = false;
while (true) {
  while (lock);
  lock = true;
  [CS];
  lock = false;
  RS;
}
```

ME X
Prog ✓
BW X
Busy waiting ✓

② Turn variable:

```
int turn = 0;
while (true) {
  while (turn != 0);
  [C.S];
  turn = 1;
}
```

```
while (true) {
  while (turn != 1);
  [C.S];
  turn = 0;
}
```

ME ✓
prog X
B.W ✓

Busy waiting ✓
deadlock free ✓

③ Peterson's soln:

```
Boolean Flag [2] = {false, false};   } 2 shared variables
int turn;
while (true) {
  Flag[0] = true;
  turn = 1;
  while (Flag[1] && turn == 1);
  [C.S];
  Flag[0] = false;
  R.S;
}
```

```
while (true) {
  Flag[1] = true;
  turn = 0;
  while (Flag[0] && turn == 0);
  [C.S];
  Flag[1] = false;
  R.S;
}
```

ME ✓
prog ✓
BW ✓
restricted to 2 process ✓
busy waiting soln ✓

④ Bakery's Algo:

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ... number[n-1]) + 1;
    choosing[i] = false;
    for (j=0; j<n; j++) {
        while (choosing[j]);
        while (number[j] != 0 && (number[j], j) < (number[i], i));
    }
        [C.S]
        number[i] = 0;
        R.S;
} while (true);
```

token →

- used for n-processes. wish to enter CS at same time.

② Hardware-type soln:

① TSL() { Returns current i/p value and sets lock = true;
- Atomic
- Non-Atomic

Boolean lock = ~~true~~ false;

```
Boolean TSL (Boolean *trg) {
    boolean rv = *trg;
    *trg = true;
    return rv;
}
```

```
while (true) {
    while (TSL(&lock);
    CS
    lock = false;
}
```

False, false;

Atomic case:	Non-atomic case:	• user-mode soln
ME ✓	ME X	• deadlock free ✓
Prog ✓	Prog ✓	• may cause starvation.
BW X	BW X	• busy waiting soln.

② swap():

Boolean key; → local (one for each process)

Boolean lock = False; → global shared variable

```
void swap (Boolean *a, Boolean *b) {
    Boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

```
while (true) {
    Boolean key;
    key = true;
    while (key == true) {
        swap (&lock, &key);
    }
    C.S
    lock = false;
    R.S;
}
```

ME ✓

Prog ✓

B.W X

③ OS-type soln:

① semaphore:

- Simple integer variable.
- It is used in mutually exclusive manner by various cooperative process to achieve synchronization.
- Integer values can be accessed using:
  - → wait() / P() / Degrade()
  - → signal() / V() / Upgrade()

} → atomic fn.

- Types of semaphore:
  - (0 to +n) Counting
    - used to control access to a resource that has multiple instance
  - (0/1) Binary
    - used to implement soln of C.S problem with multiple processes.

Critical Section Sol^n using Semaphore:

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

```
signal(S){
    S++;
}
```

Critical Section Sol^n using busy waiting:

```
wait(S){
    S--;
    if(S<0){
    // add process to queue
    block();
}}
```

```
signal(S){
    S++;
    if(S<=0){
    // remove process from queue
    wakeup();
}}
```

What semaphore does?

```
while(true){
    wait(S);   ES
    (S);
    signal(S); ES
}
```

if S=3 } 3 processes can enter C.S at same time.

if Bin. Sema. } S=1 ⇒ signal(S) ⇒ S=1

characteristics:

- used to provide ME.
- control access to resources.
- can have deadlock, starvation, busy waiting problem
- Also leads to priority inversion.
- Machine Independent.

Classical problem of synchronization

① Bounded-buffer problem / producer-consumer problem

② Reader-writer problem.

③ Dining philosopher problem.

① Producer consumer:

producer:

```
wait(empty);   (deadlock) when buffer is full ↕ swap
wait(mutex);
// add item
signal(mutex);
signal(full);
```

consumer:

```
wait(full);    (deadlock) when buffer is empty. ↕ swap
wait(mutex);
// consume item
signal(mutex);
signal(empty);
```

Swap → No deadlock.

Variable initialization:

- buffer size = n.
- Mutex = 1; } BS
- Full = 0; } CS
- Empty = n;

② Reader - writer :

writer	reader
wait (wrt);	wait (mutex);
// write	readcount ++;
signal (wrt);	if (readcount == 1)
	wait (wrt);
	signal (mutex);
	// Reading
	wait (mutex);
	readcount --;
	if (readcount == 0)
	signal (wrt);
	signal (mutex);

BS { • Mutex = 1 → for accessing readcount
• wrt = 1

CS • readcount = 0

if removed. → only 1 reader will be allowed at a time.

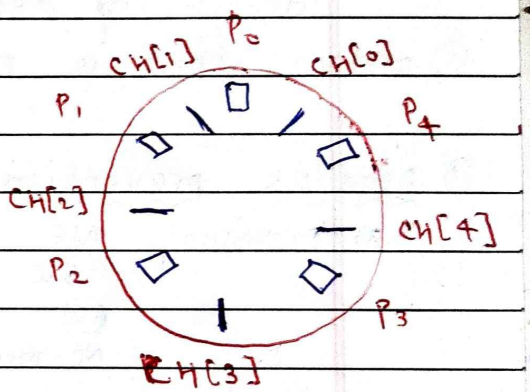
	Reader	writer
Write	✗	✗
Read	✓	✗

③ Dining - philosopher :

• k - philosopher
• k - chopstick.
• Can lead to deadlock !

Soln :
① (k-1) philosopher
k chopstick.
② if both chopstick available then eat else not
③ one must pick left chopstick first then right one, other same
↳ (avoid circular wait)

(Pi)
wait (CH[i]);
wait (CH[(i+1) mod k]);
// Pi eating.
signal (CH[i]);
signal (CH[(i+1) mod k])



Operations on Resources :
1. Request
2. Use
3. Release

necessary cond^n for deadlock :
① ME ② Hold & wait
③ No - preemption
④ circular wait.

koi bhi ek unsatisfy hoga to deadlock nahi hoga

Deadlock : If two or more process are waiting for cond^n which is never going to occur.

Resource Allocation graph:

Vertices:
- Process ◯
- Resources [...]

Edges:
- Request: from 'p' to Resour
- Allocation: from Instance to 'p'.

Deadlock soln:

① Make sure deadlock never occurs.
- deadlock prevention (Not practical)
- deadlock avoidance (Banker's Algo)

② Allow deadlock to occur, then detect & recovery.

③ pretend there is no deadlock.

① Deadlock prevention:

a) Preventing ME, H&W, ~~RW~~, No-preem } any one ~~to stop~~ deadlock.

Note: dA less | dP more restrictive

② Deadlock avoidance: In this, request for any resource will be granted if the resulting state of system doesn't cause deadlock in system.

- Banker's Algo: $Need_i = Max_i - Alloc_i$

Process	Allocⁿ	Max	Needi
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

Available:

3 3 2 → initially.
5 3 2 after P_1
7 4 3 P_3
7 5 3 P_0
10 5 5 P_2
10 5 7 P_4.

System is in safe state

- safe sequence: $< P_1, P_3, P_0, P_2, P_4 >$
- More than one safe seq is possible
- max^m: n! (n = process)

Resource Allocatⁿ Algo:

If $Req_1 = <1, 0, 2>$ then

<Algo>
① if $Req_i \leq Need_i$
② if $Req_i \leq Available_i$
③ $Alloc_i = Alloc_i + Req_i$
$Need_i = Need_i - Req_i$
$Avail = Avail - Req_i$
④ Run safety algo. (Bankers Algo).
- safe state → accept
- unsafe state → reject

system → 3 process, 4 instance
Max. req = k instance

∴ Max res.

A	k
B	k
C	k

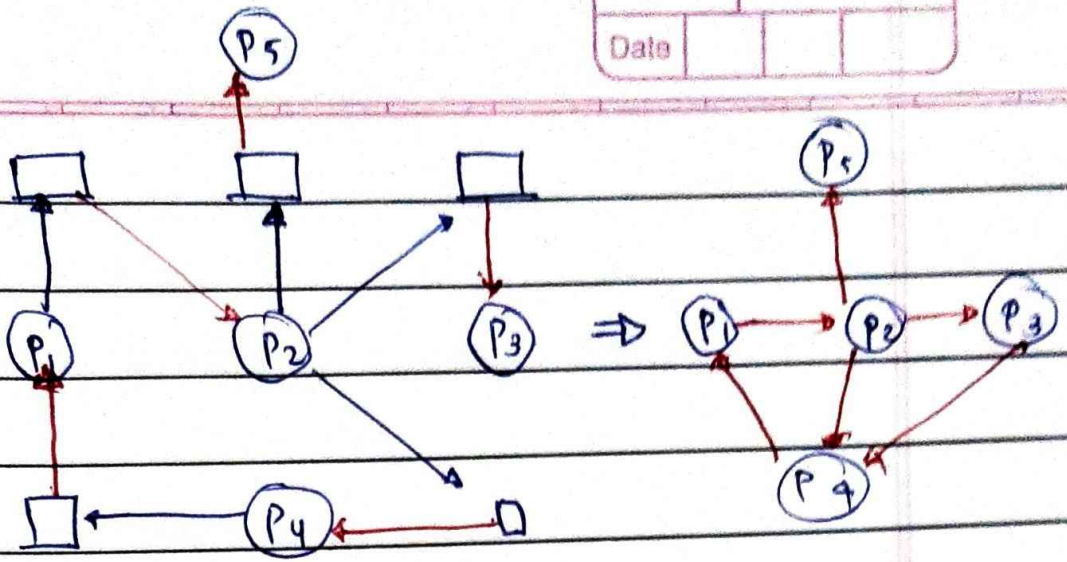
$3(k-1) + 1 \leq 4$.
$k \leq 2$ ✓

(3) Deadlock Detection:

case1: When all resource have single instance

↓

If cycle is present ⇒ Sure deadlock.



< Wait-for-graph >

case2: When resource have Multiple instance

↓

presence of cycle ⇒ possibility of deadlock.

Use this algo ⇒ If all process finish → No deadlock.

Process	Alloc	Req.
P_0	010	000
P_1	200	202
P_2	303	000
P_3	211	100
P_4	002	002

Avail	
000	→ initially assume
010	after P_0
313	P_2
513	P_1
724	P_3
726	P_4.

Memory Management: (MM)

→ process of controlling & coordinating with main memory

- Functions:
  - mem allocat^n
  - mem deallocat^n
  - mem protect^n
- Goals:
  - → Max^m utilizat^n of space
    - ↳ min. fragmentation
  - → Ability to run larger prog. with limited space

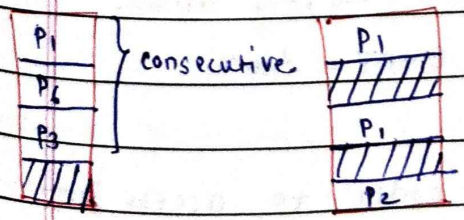
MM technique:

→ To store processes in main mem.

We have 2 technique:

Contiguous	Non-contiguous
• Entire process stored in a contiguous/consecutive memory location.	• Entire process stored in memory in scattered form.

- Contiguous
  - Fixed partition → int frag
  - Variable partition → ext frag
- Non-cont
  - paging → int frag
  - segmentation → ext frag



① Fixed partition:

- Entire mem. is divided into fixed no. of partition.
- size of each partition may variable.
- Each partition can accomadate only 1 process at a time.
- ~~degree of partition = No of partition~~

② Variable Partition:

- A new partition is created at runtime, equal to size of new process.

Internal frag:

→ If extra space is allocated to a process which is more than req^d space of process.

External frag:

→ In this, we have free mem. blocks but we cannot assign it to process.

Sol^n → compaction (rearrangement of mem blocks)
→ paging

Partition Allocation policy:

→ For selecting partition among available partitions.

200 MB
100 MB
150 MB
50 MB
300 MB

process size	(P1) 150 MB	(P2) 125 MB	(P3) 50 MB.
① First fit	200	150	100
② Best fit	150	200	50
③ Worst fit	300	200	150
④ Next fit	200	150	50

First fit: search from start

Next fit: search from last allocation.

Paging:

→ process is divided into equal size of partition.

→ physical mem is divided into same equal size of frame

∴ (page size = frame size)

→ pages are scattered in frames

→ page table is used to map pages & frames.

→ PT is also stored in frames (MM).

→ If PT → very small, the stored in register

→ Each process has its own PT.

→ No. of entries = No. of pages in PT in process.

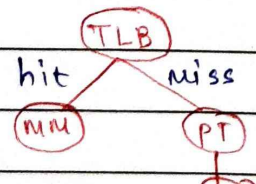
→ PTBR: stores base addr. of PT.

TLB (Translation lookaside buffer):

→ used to improve paging performance.

→ small & fast mem. hardware used to store frequently used PT entries.

H = hit
1-H = miss



• EMAT =

$H(t_{TLB} + t_{mm}) + (1-H)(t_{TLB} + 2t_{mm})$

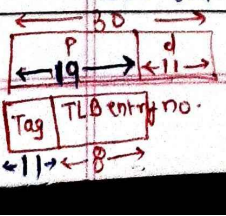
(PT + MM)

TLB mapping: To store PT entries in TLB in order to access faster.

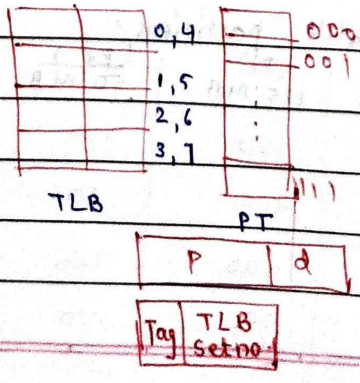
① Fully associative:

Any PT entry can be at any place in TLB.

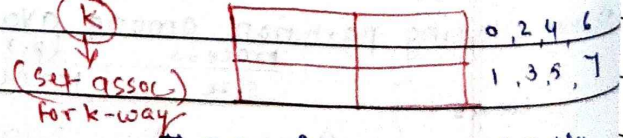
qn) LA = 30 bits
page size = 2KB = 2^{11} bits
TLB size = 256 entries = 2^8 entries



② Direct mapping:



③ Set associative mapping:



sets in TLB = $\frac{\text{no. of entries in TLB}}{k}$

TLB set no. = $\log_2$ (no. of sets in TLB)

formulas:

① logical addr:

byte no. $= \lceil \log_2 (\frac{\text{page}}{\text{size}}) \rceil$

no. of bits req^d to generate total pages $= \lceil \log_2 (\text{no. of pages in process}) \rceil$

• no. of pages $= \frac{\text{process size}}{\text{page size}}$

② physical addr:

no. of bits $= \lceil \log_2 (\frac{\text{page}}{\text{size}}) \rceil$

no. of bits $= \lceil \log_2 (\text{no. of frames in MM}) \rceil$

• no. of frames in MM $= \frac{\text{M.M size}}{\text{page size}}$

③ page table:

• PT size = # pages in process × 1 PT entry size

= # pages in process × (frame no. bits + extra bits)

④ Time req^d:

• EMAT = 2 tmm

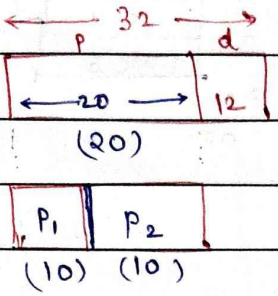
If PT → very small,

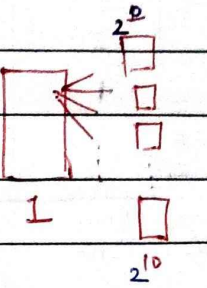
• EMAT = tmm

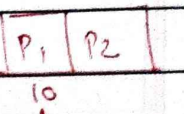
Multilevel paging:

- when entire PT cannot fit into a single frame, hence PT is also distributed over multiple frames.

eg. LA = 32bit
page size = 4kB
PT entry size = 4B
No. of levels? (2)







P1 searching in P2 with 10 entries

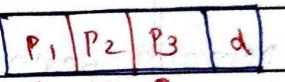
→ No. of PT entries in single page = page size / PT entry size = 4kB / 4B = 2^{10}

pages in outermost PT = 1 (always)

pages in inner PT = 2^{10}

1025 pages req.

IIIly if



pages in outermost = 1

pages in inner = 2^8

pages in innermost PT = $2^8 \times 2^8$

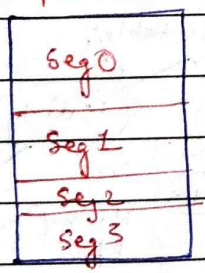
1029 pages.

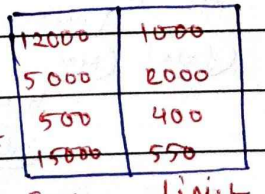
Segmentation:

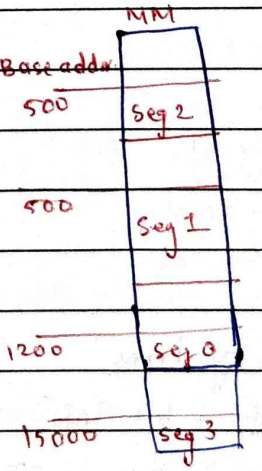
→ process is divided into segments of variable size and are scattered in phy. mem. (MM).

LA: | s | d |

s → no. of segment







optimal page size

∴ Total no. of pages req^d = $\frac{S}{P}$

∴ Total PT size = $\frac{S}{P} \times D$

∴ Avg. interal frag = $\frac{0+P}{2} = \frac{P}{2}$

∴ Total overhead = $O = \frac{SD}{P} + \frac{P}{2}$

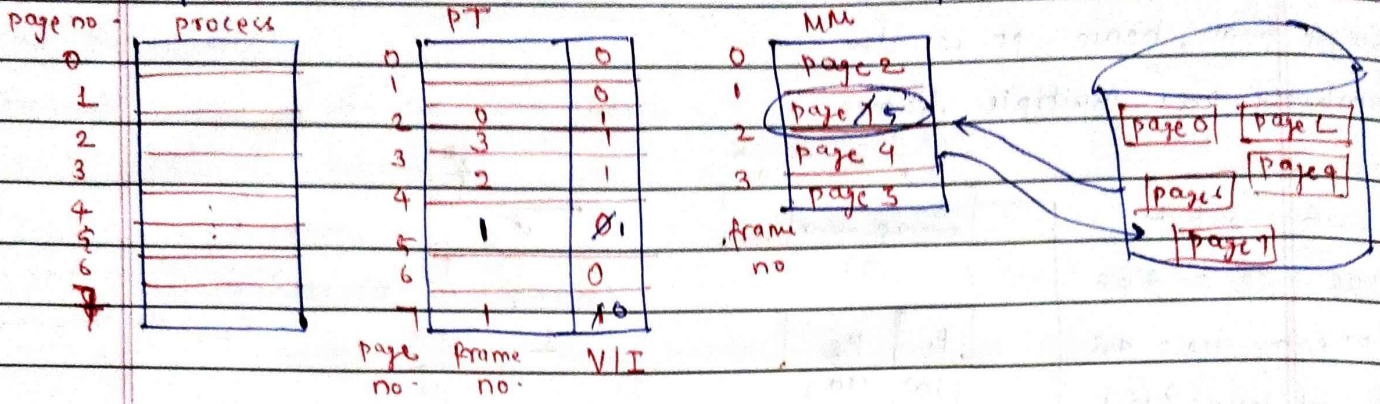
$\therefore \frac{dO}{dP} = \frac{1}{2} - \frac{SD}{P^2} = 0$

let,
process size = 'S' Bytes
col. width of PT = 'D' Bytes
optimal page size = 'P' Bytes

$\therefore P = \sqrt{2SD}$.

virtual memory:

- feature of OS that enables to run larger process in smaller memory.



Demand paging.

→ Bring pages from SM to MM when CPU demand.

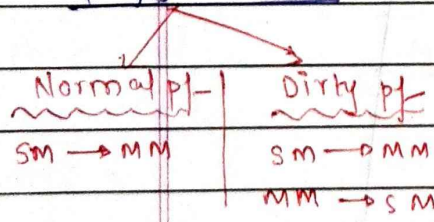
→ (page fault (pf)) ⇒ when demanded page is not available in MM.

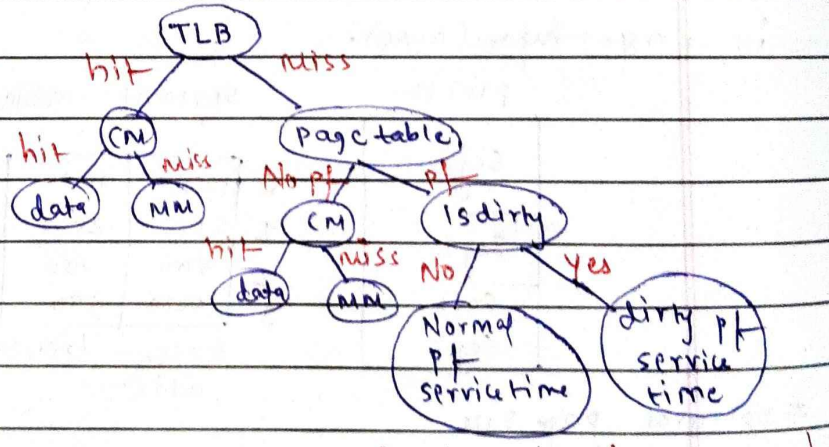
→ once page fault occurs, OS performs its service in page fault service time

→ In (pfst): faulted page moved from SM → MM

If some space already available in MM, then faulted page is stored there.
else some one page is replaced and instructions restarted again
~~it doesn't resume.~~

- page fault

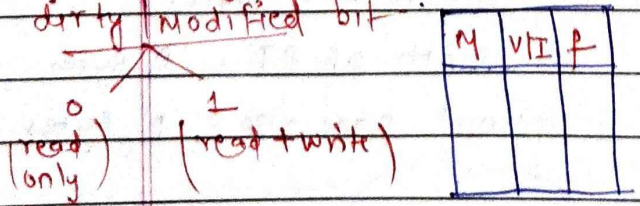




page write polity time:

→ It can be saved if 1 bit info^n is maintained for each page in PT

dirty / Modified bit



→ only dirty pages are written back to sec. MM after replacement.

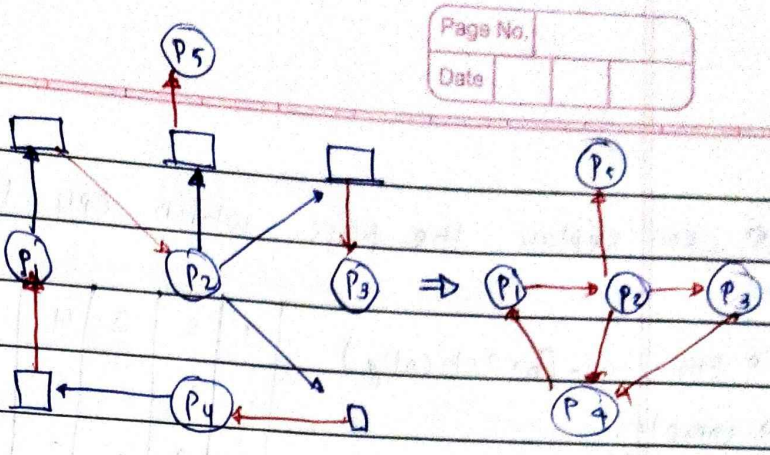
Note

EMAT = H × (t_TLB + t_MM) + (1−H) [t_TLB + t_MM + (1−p) t_mm + p × (page transfer time from SM → MM + d × page transfer time from MM → SM)]

③ Deadlock Detection:

case1: When all resource have single instance

If cycle is present ⇒ Sure deadlock.



<Wait-for-graph>

case2: When resource have Multiple instance

presence of cycle ⇒ possibility of deadlock.

use this algo: If all process finished → No deadlock.

Process	Alloc	Req.	Avail	
P0	010	000	000	initially assume
P1	200	202	010	after P0
P2	303	000	313	P2
P3	211	100	513	P1
P4	002	002	724	P3
			726	P4.

Page Replacement policy:

✓ → page faults.

X → page hit

① FIFO:

Replace that page which is brough to MM first:

- easy to implement
- low overhead
- more page fault (poor performance)
- Belady's anomaly problem.

page fault rate = 7/12

	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
	✓	✓	✓	✓	✓	✓	✓	X	X	✓	✓	X

Belady Anomaly → # frame ↑ ⇒ # page faults ↑ in some cases in FIFO only.

② Optimal policy: Replace place which is not going to be referred in near future:

- easy to implement
- highly efficient (not practical)
- Time-consuming

page fault rate = 7/12

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	3	3	3
	2	2	2	2	2	2	2	2	2	4	4
		3	4	4	4	5	5	5	5	5	5
✓	✓	✓	✓	X	X	✓	X	X	✓	✓	X

③ LRU: Replace the page which CPU has not used since longest time.

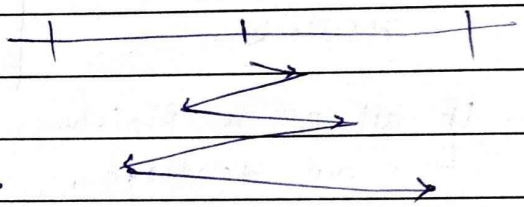
- Efficient (practically)
- complex
- expensive

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	3	3	3
	2	2	2	1	1	1	1	1	1	4	4
		3	3	3	2	2	2	2	2	2	5
✓	✓	✓	✓	✓	✓	✓	x	x	✓	✓	✓

Disc scheduling Algo: done by OS to schedule I/O requests arriving for disk

① Fcfs: Req. are serve in order they arrive in disk queue.

- Every req. gets a fair chance.
- Not provide optimize seek time.

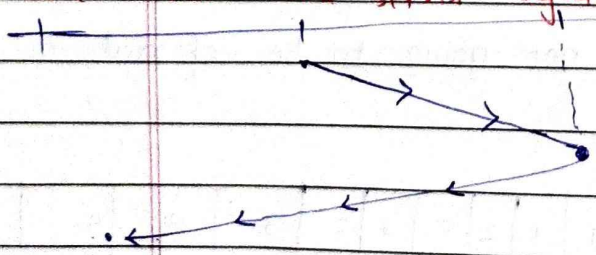


Total head movement = Total seek time

② SSTF: Req. having shortest seek time executed First

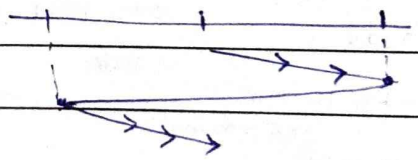
- Avg RT ↓ throughput ↑
- starvation possible

③ Scan (Elevator): disk head moves in a particular direct^n & completes servicing req. till end of disk, then reverse the direct^n and serves again.



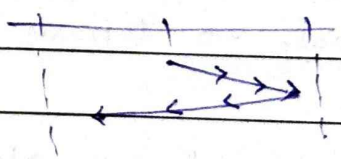
- Throughput ↑ Avg RT ↓
- long WT for ~~process~~ req. for recently visited location

④ C-scan: disk head moves in circular direct^n.

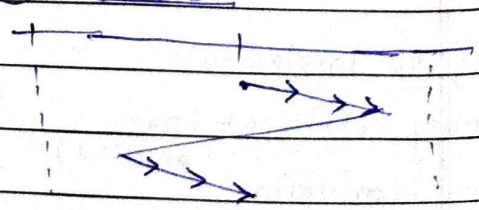


More uniform WT.

⑤ look:



⑥ C-look:



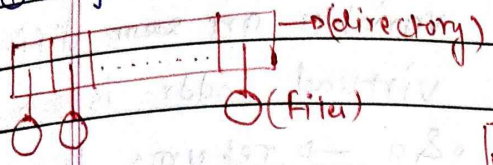
File system:

- File directory (Folder):

→ collectn of files.

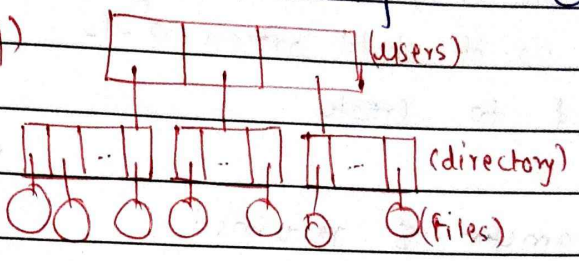
- Types:

① Single level directory: → single directory for all users.

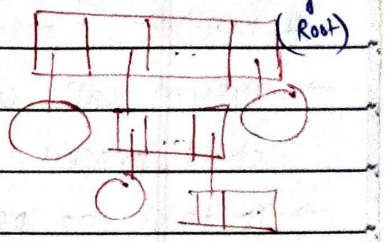


- Naming problem
- grouping problem

② Two-level directory: → seperate directory for each user



③ Tree-level directory



- File allocation method: (non-cont)

① Contiguous:

→ single contiguous set of block allocated to a file at a time of allocatin.

- External frag
- ~~Compactn needed to free disk.~~
- It allows Random access of block.

② linked:

→ no. ext frag.

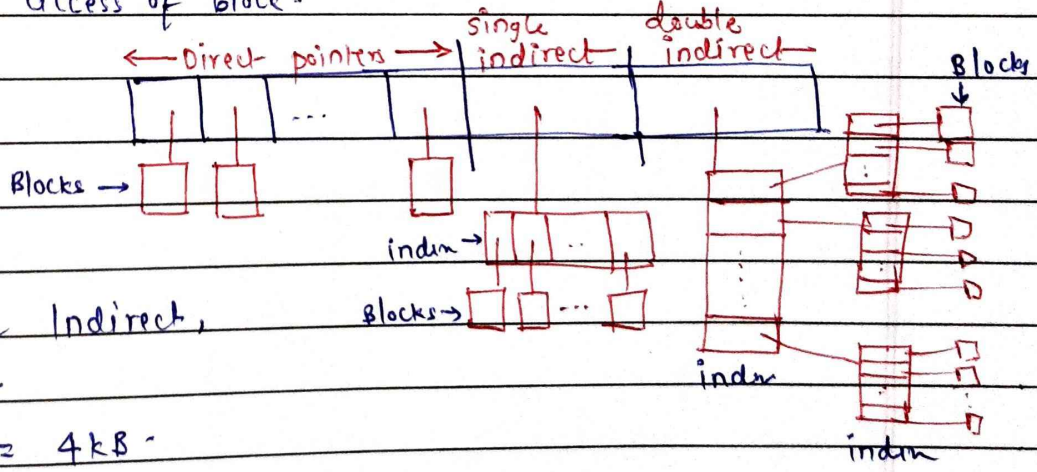
- Internal frag.
- only sequence access is allowed.

③ Indexed.

→ allows seq. & direct access.

→ no. ext frag.

Unix I-Node:



eg. 12-direct, 1 single Indirect, 1 double Indirect

disk block size = 4kB.

disk block addr = 32 bits long

Max^M possible file size?

→

12	1	1
direct	single indirect	double indirect

$12 \times 4kB + 1k \times 4kB + 1k \times 1k \times 4kB$

$= 4GB$

No. of Indexes per disk block $= \frac{\text{Block size}}{4B}$

$= \frac{4kB}{4B} = 1k$ ✓

System calls:

→ programmatic way in which a comp. prog. requests a service from kernel.

fork()

- Total no. of process = 2^n
- Total no. of child process = $2^n - 1$.

→ System call used to create child process.

→ It takes no parameter & returns:

-ve value: child process creat^n was successful.

zero: Returned newly created child proces

+ve value: Returned value contains processID of newly created child process

- physical addr. of both variables are ~~same~~ diff. but virtual addr. is same.
- &a → returns the virtual address.