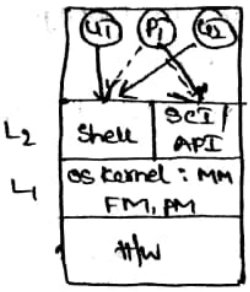
Operating systems :

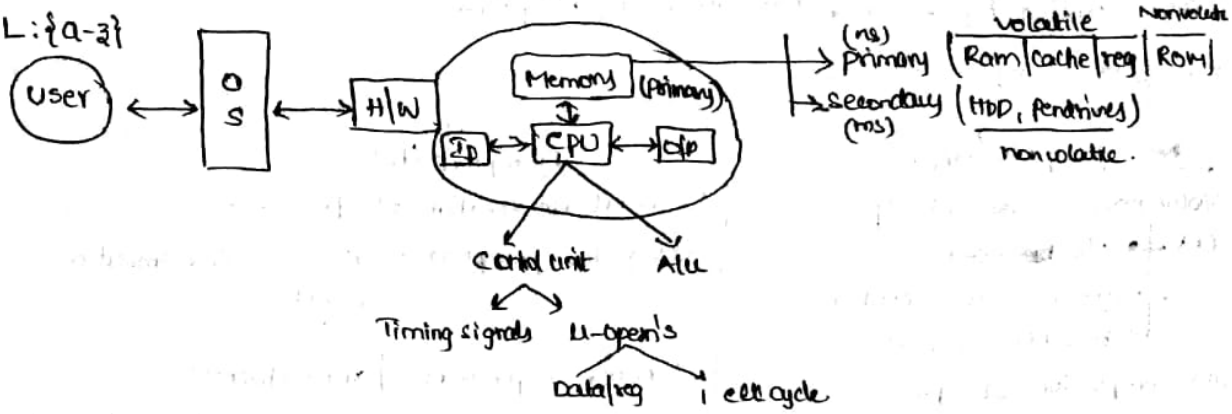
OS :

→ Interface b/w user & H/W
→ Collection of pgms that control system
→ Resource manager (H/W & S/W)
→ Set of utilities to simplify Application development
→ Acts like a Govt.

OS :



*) Von-neumann Arch (stored pgm Concept of Computation)
↳ loaded into mem for Execution



Functions and Goals of OS

Goals : (CEPSRR)

→ Convinience (Easy to use)
→ Efficiency (utilization, managing)
→ portability (Run on any platform)
→ scalability (Accept changes)
→ Reliability (state should be always correct)
→ Robustness (Small Error shouldn't break os)

*) Primary Goals may change A/c to domains.

→ personal Computing : Convinience (Primary)
Efficiency (secondary)
→ Real time systems : Efficiency
Hard (space, Aircraft) soft (washing m/c's ATM's) Reliability Robustness.
→ Smart phone Computing : power efficiency

Functions :

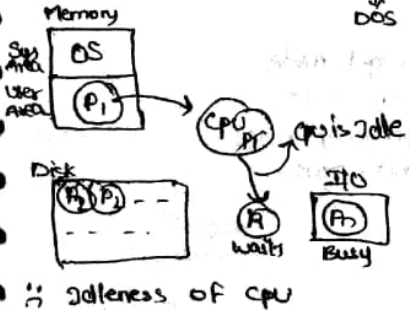
→ Device mgmt
→ Memory mgmt
→ process mgmt
→ File mgmt

Spooled device : A device for which tasks/jobs are placed in Queue i.e., job Queue

Types of OS :

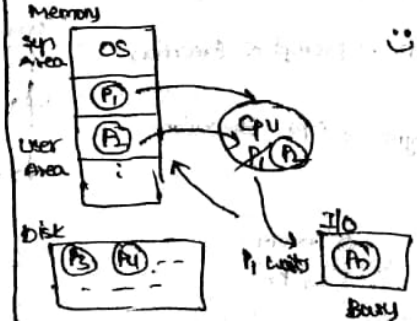
Based on Disk technology

uni-programmed os (Ex: Batchos ↓ DOS)



∴ Idleness of CPU

Multi programmed os : (UNIX, LINUX, WINDOWS, MAC)



∴ Max CPU utilization ie- Throughput
↓
(Num of jobs completed Per unit time)

Definition of Multiprogramming :

→ Ability of os to manage, ~~mem~~ multiple ready to run programs in memory is called Multiprogramming.

→ Multiplexing of cpu among ready programs in memory is called multiprogramming

Types of Mp os :

User's view : OS

- **Single user os**
  → All the pgms in the memory are submitted by one user
- **Multi user os**
  → All the pgms in the mem are submitted by many users

Based on premption : OS

- **Non-premptive (NPr)**
  Voluntary release of CPU
  (i) → I/o ~~req~~ need
  → system call Execution
  → resource request
  (ii) Completion of process.
  ∵ Starvation
  Poor responsiveness | Interactiveness
  Ex:- Win 3.0
- **Premptive (Pr)**
  Forceful deallocation of process.
  Criteria for premption : Time – Time shared os
  Priority
  :
  ∵ better responsiveness | Interactiveness.
  Ex:- Win 95--10, UNIX, LINUX

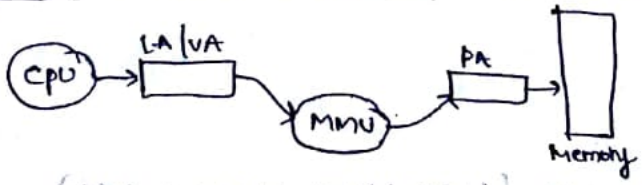
Architectural requirements for Mpr OS Implementation :

① I/O (secondary storage) : DMA capability should be there.

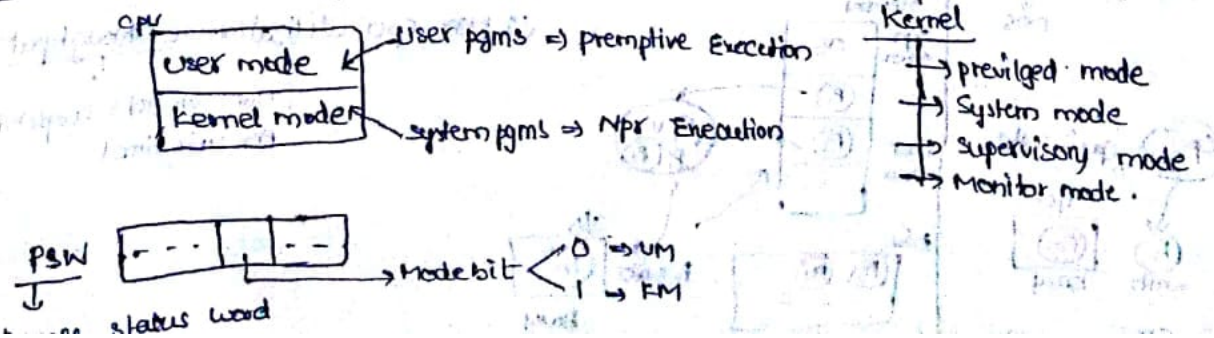
Mode of transfer (I/O)

- Pgm Controlled IO → Polling → ✗ Mpr
- Interrupt driven IO → ✓ Mpr ∵ processor load ↑
- DMA → ✓ Mpr ∵ processor load ↓

② Memory : Address translation read because it provides security



③ CPU : Dual mode operation read



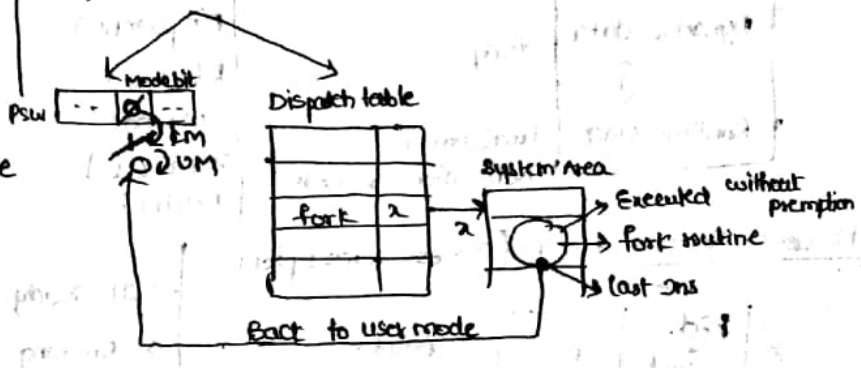
Mode shifting :

BSA : Branch & Save Addr
SVC : Supervisory call
ISR : Interupt Service routine

```
main()
{
  int a,b,c;
  a=1; b=2;
U c= f(a,b);
L printf("%d",c);
S fork();
}
```

At Compile time → BSA, SVC (↑ opcodes)
At runtime → s/w interupt (ISR)

*) User mode —Interupt→ Kernel mode
Kernel mode —Not read→ User mode



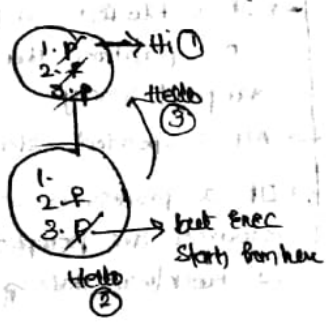
*) Interupts
- h/w (raised by devices)
- s/w (pgm driven)

functions

- User defined — f(a,b)
- Library — printf( )
- System call (defined in OS) — fork()

fork system Call :

```
main()
{
1. pf("Hi");
2. fork();
3. pf("Hello");
}
```



*) Num of forks in series	Num of Child	Total Processes
1 fork	1	2
2 fork	3	4
3 fork	7	8
n fork	$2^n - 1$	2^n

Return value of sys call

- -ve (-1) — Error failure
- 0 — Success — Child
- +ve (>0) — Parent

```
main()
{
 // parent //
 ret = fork();
 if(ret == 0)
 {
   // child
 }
 else
 {
   // parent
 }
 // both
}
```

Child Executes if Parent Executes Else

*) Current state of values are copied into child process.

Process Concepts :

(Passive Entity) (Active Entity)

Program vs process :

(.exe) (pgm under Execution)

using of resources like cpu, IO, memory

- Instructions
- Data
  - Static
    - fixed size
    - Load time
  - Dynamic
    - var size
    - Run time

Definitions of Process

- pgm under Execution
- Instance of a pgm
- Active Entity
- schedulable unit
- present in memory
- Unit of cpu utilization
- locus of control of OS
- Animated spirit of pgm

Developers view :

Process can be viewed as ADT

<Def ; Rep ; opers ; Attr>

Imp

Representation

Memory

Instructions	Code / Text section
(static) data	← Global var ← static storage class
Dynamic data ↕ Runtime stack	heap local var + Activation records

Opers :

*) PCB is involved from creation to deletion

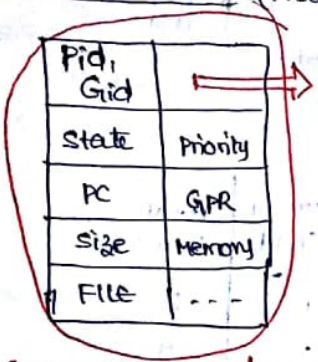
- Create ( )
- Schedule ( )
- Dispatch ( )
- Block ( )
- Suspend ( )
- Resume ( )
- Delete ( )

Attributes

*) Pid, Parentid, Groupid, PC, GPR, Type, State } CPU Attr

*) Size(B), Memory limits, list of open files, list of open devices } Memory Attr

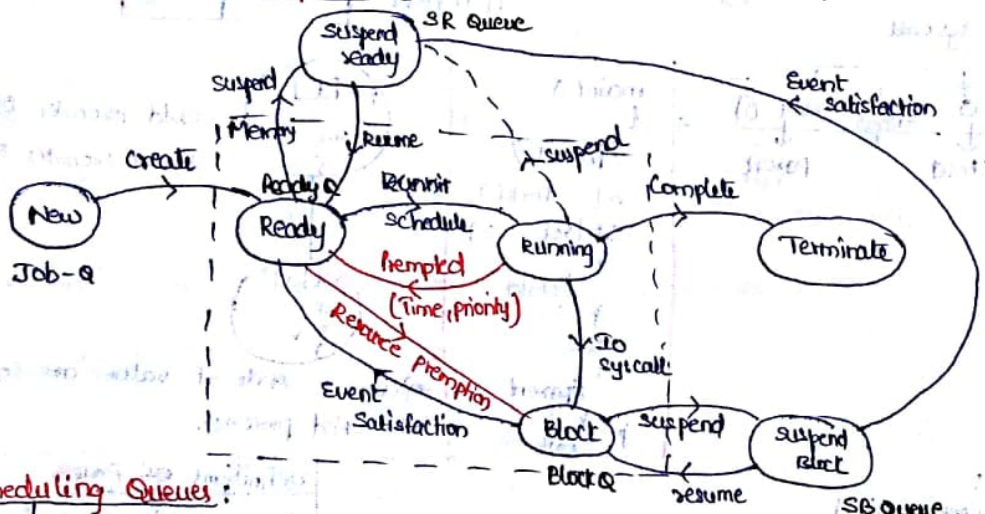
Process Control Block (Process descriptor)



Context of process / Process Environment

- If ready not present ⇒ uniprogrammed OS
- Running → ready ⇒ prempted OS
- Running ↛ ready ⇒ Nonprempted OS
- If a Blocked suspended process completes its IO operation then the proces would go to suspend ready.
- All suspended states are present in sec memory.
- If a process is in ready state and one of its resources are prempted then the proces would go to block/wait state.

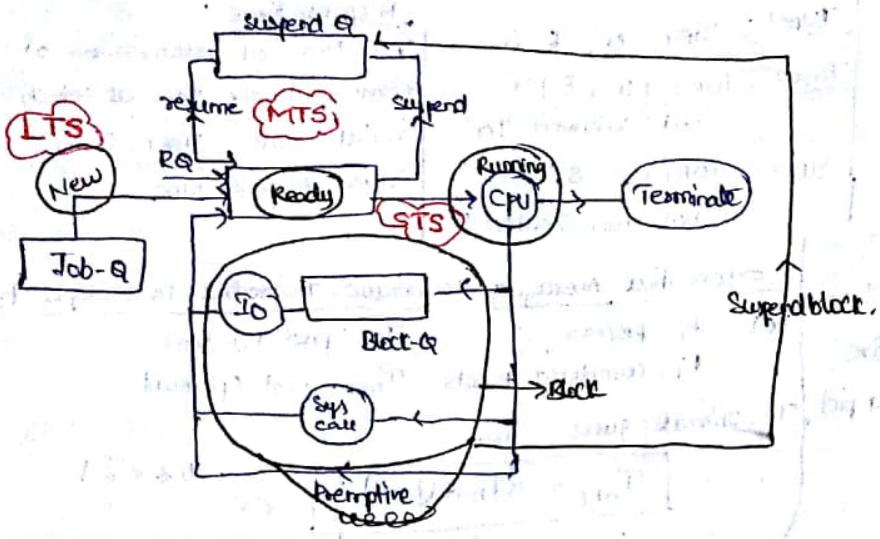
State diagram of a process (DFA)



Scheduling Queues :

- In memory
  - Ready Q (list of PCB's): RQ → Pi Pj PK
  - RQ → PCBi → PCBj → PCBk
  - Block Q: Devices
    - DB1 → Pa Pb Pc : D1
    - DB2 → Pd Pe : D2
    - DB3 → Pf Pg : D3
- On-disk
  - Suspend Q → list of suspended Pgms
  - Input Q / Job Q → Pgms ready to get loaded

State Queuing diagram :



Schedulers & Dispatchers :

Scheduler

- **LTS** — Controls degree of multiprogramming
- **MTS** — → supersions & resumption → swapper
- **STS** → cpu schedule

Dispatcher :

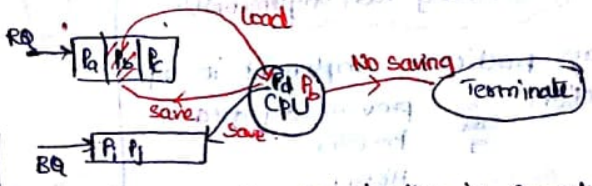
→ carry out activity of **Context switching**

- Saving the PCB of prempted process and loading the PCB of next process to run on CPU.

Context switching time = time dispatch latency

↓

Overhead.



*) save is a option but loading is compulsion

Cpu scheduling : (STS)

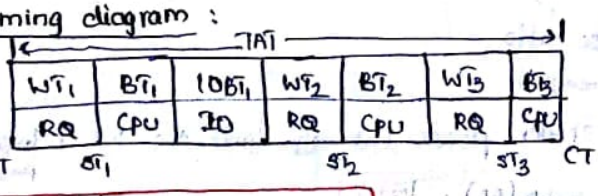
Function : select a ready process for running in cpu

Goals : Max cpu utilization (Throughput)
Min Turn Around time
Min Waiting time → starvation, → response time.

Process times :

1. Arrival time
2. Waiting time
3. scheduling time
4. Burst time
5. IO burst time
6. Completion time
7. Turn around time

Timing diagram :



$TAT = CT - AT$

$WT = TAT - [BT + IOBT]$

*) $TAT(P_i) = C_i - A_i$

$\text{Avg TAT} = \frac{1}{n} \Sigma (C_i - A_i)$

*) $WT(P_i) = (C_i - A_i) - (BT_i + IOBT_i)$

$\text{Avg WT} = \frac{1}{n} \Sigma (C_i - A_i - (BT_i + IOBT_i))$

*) Schedule length (L) = $MAX(C_i) - Min(A_i)$

*) Schedulable combinations.

Non premptive – $n!$

premptive – ∞

*) Throughput (μ) = $\frac{n}{L}$ i.e. $n \rightarrow L$, $2 \leftarrow 1$

Cpu scheduling Techniques :

① FCFS :

Criteria : AT
Mode of open : Non pemptive.
Conflict resol : lower pid.

Type1 : IOBT = 0, S = 0
Type2 : IOBT ≠ 0, S ≠ 0 but Concurrent IO
Type3 : IOBT ≠ 0, S ≠ 0 but non-Concurrent IO

Response time :
The time of submission of a request to the time at which the Initial results, start Coming is called Response time

② SJF :

Optimal Algo.

Criteria : BT
Mode of open : Non pemptive
Conflict resol : AT + lower pid

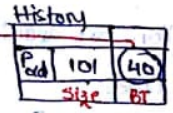
Performance of SJF :

favours shorter process.

∵ Max Throughput, Min TAT, WT (Avg) | ∵ Starvation to larger processes

Practical Limitation :
Before running we should know BT
Practically Non Implementable

We can practically Implement it by Prediction → done by prev History data of Processes



Exponential Averaging Technique to predict next CPU BT :

let P_i : process, T_i : predicted burst
t_i : completed Bursts T_{n+1} : next cpu burst
T_1 : Initial guess Then.

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n \quad \rightarrow (1) \qquad 0 \leq \alpha \leq 1$$

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

$$(1) \Rightarrow T_{n+1} = \alpha t_n + (1-\alpha)\left[\alpha t_{n-1} + (1-\alpha) T_{n-1}\right]$$

$$= \alpha t_n + \alpha(1-\alpha) t_{n-1} + (1-\alpha)^2 T_{n-1}$$

$$\vdots$$

$$T_{n+1} = \alpha t_n + \alpha(1-\alpha) t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + \cdots \quad : T_1$$

Exponential Averaging.

③ SRTF : (Premptive SJF)

Criteria : BT
Mode of open : premptive
Conflict resol : lower pid

If premption is done
SRTF > SJF (Powerful)
If premption is not done
SRTF ≡ SJF

Practically non Implementable

④ LRTF :

Criteria : BT
Mode of open : premptive
Conflict resol : lower pid

⑤ HRRN :

Criteria : Response ratio
Mode : Non premptive
" Favours not only shorter process but also limit the WT of longer process.

$$\text{Response ratio (RR)} = \frac{W+S}{S}$$

⑥ priority based scheduling :

Criteria : priority (Integer value)
static | dynamic

Mode : Nonpr | Pr

*) Priority : 10 (L) 100 1000 (H)

⑦ Round Robin scheduling : (Premptive FCFS) (Time shared os).

Criteria : AT + TQ
Mode : premptive

Performance of Round Robin:

Value of TQ

Small	very small	large	very large
→ More Context switch o/h → Improved responsiveness	→ cpu is busy in Context switches rather than doing useful work	→ less Context switch o/h → less responsiveness	→ Similar to FCFS.

Points to remember:

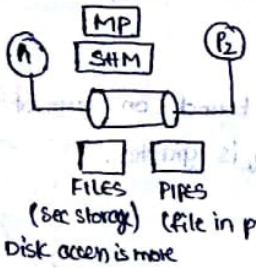
	Technique	Mechanism	Criteria	Mode of oper	Conflict resol
①	FCFS	First Come First serve	AT	Npr	lower pid
②	SJF (optimal)	Shortest Job First	BT	Npr	AT + lower pid
③	SRTF	shortest remaining time first	BT (Non Implement)	Pr	lower pid
④	LRTF	longest remaining time first	BT	Pr	lower pid
⑤	HRRN	Highest response ratio next $= \frac{w+s}{s}$	RR	Npr	lower pid
⑥	Priority	Priority based scheduling	(same type) Priority (High value)	Npr / Pr	Priority + lower pid
⑦	Round Robin	Time quantum	AT + TQ	Pr	lower pid (If TQ is very big)

Process management:

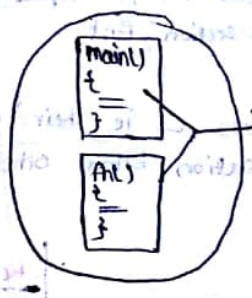
IPC; synchronization; Concurrency:

Need for synchronization: Inconsistency [wrong results]
Data loss [By speed, By heavy Context switch o/h's] → Process Competetion
Deadlocks → Process Co-operation

Inter process Communication:



Intra process Communication:



Comm done through
1) Global var
2) parameter passing

Process Synchronization: — Competetion / Cooperation

Process Competetion:

Two or more processes are said to be in Competetive synchronization if they compete for the accesibility of shared resource

→ lack of sync among Competing process leads to **Inconsistency or Data loss**

Process Co-operation:

If they get affected by Each other

→ lack of sync among Cooperating process leads to **deadlocks**

*) An Application in IPC Environment may Involve Either Competetion/Co-operation/Both type of Synchronization.

Synchronization Mechanisms :

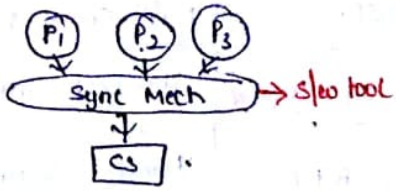
Neccesary Conditions :

① **Critical section** : (It is the part of the pgm where shared resources are accred & manipulated)

② **Race Condition** : The situation where multiple processes tries to access and manipulate the critical resources concurrently and the outcome depends on the order in which they finish their update.

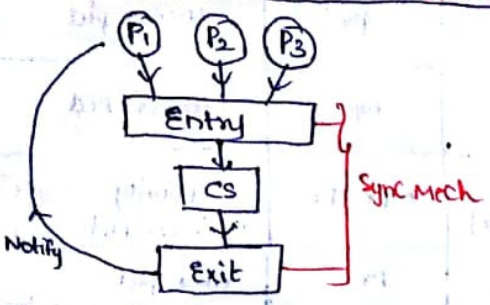
③ **Premption** : Root cause for lack of sync

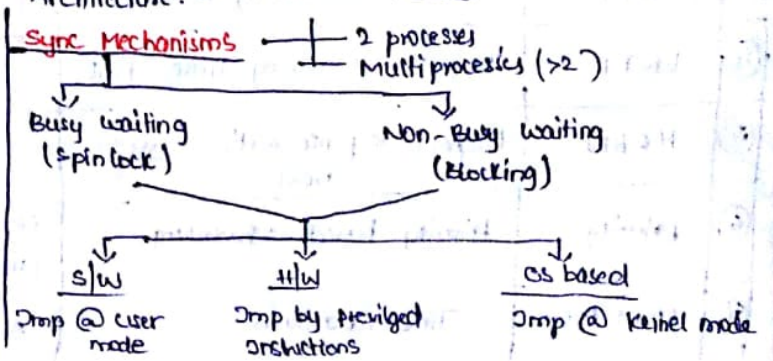
*) All 3 conditions are suffitient for sync problem → sol



Synchronization Mechanism Architecture :

Architecture.





Requirements for sync Mech :

① **Mutual Exclusion** :

→ Mandatory requirement

→ No two processes may be simultaneously present in CS.

② **Progress** :

→ No process running outside the CS ie (NCS) should block the other Interested process from Entering CS.

→ In other words a process in NCS should not participate in the decision making of Interested Processes as to who should Enter Critical Section first.

③ **Bounded Waiting** :

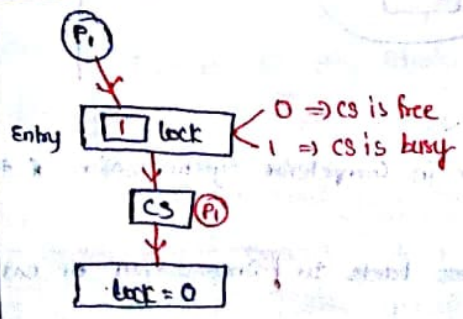
→ No process has to wait forever to access CS ie their must be a bound on num of times a process is allowed to Enter Critical section before other process req is granted.

Lock variable sync Mech :

→ Busy waiting

→ s/w solution

→ Multiprocess Mech



Result

ME : ×

Progress : ✓

Bounded waiting : ×

Implementation : (HLL)

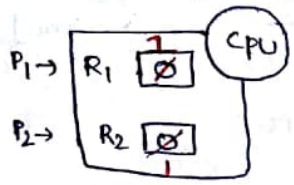
```
int lock = 0;
{
  while(1)
  {
    a) Non_cs();
    b) while ( lock != 0 );   (Entry)   (JNZ)
    c) lock = 1;
    d) <cs>
    e) lock = 0;              (Exit)
  }
}
```

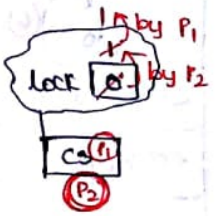
(LLL)

```
int lock = 0;
{
while(1)
{
  a) NON_CS;
  b) Load Ri, LOCK
  c) CMP Ri #0
  d) JNZ step b
  e) store lock #1            (Entry)
  f) <CS>
  g) Store lock #0.           (Exit.)
}
```

Mutual Exclusion : (X)

RQ | P1 | P2 | P3





(Low level long)

t_1 : (P1) : a; b; c; d; pre.

t_2 : (P2) : a; b; c; d; e; f; pre.

t_3 : (P1) : e; f

ME ✗

Progress : (✓)

The process running in Non cs doesn't affect the Interested process.

Bounded waiting :

A process after using cs again make lock = 1 and Enter into cs which makes another Process to starvate

(2) Strict Alternation Sync Mech :

- Busy waiting
- s/w solution
- 2 process sol

Result :

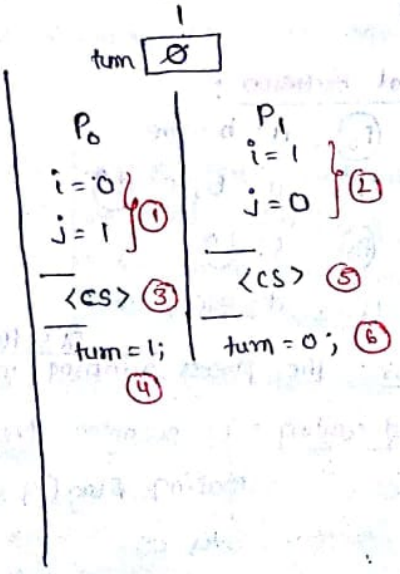
ME : ✓
Progress : ✗
Bounded waiting : ✓

Implementation :

```
int turn = rand(i,j);
void process (int i)
{
  int j = not i;
  while(1)
  {
    a) NON_CS();
    b) while (turn != i) ;    (Entry)
    c) <CS>
    d) turn = j;              (Exit)
  }
}
```

low level

```
Entry.
a) Load R1, turn
b) Load Rj, i ;
c) Comp R1, Rj
d) Jump to @
e) <CS>
f) Load R1, turn
g) Load Ri, j
h) Store R1, Rj
```



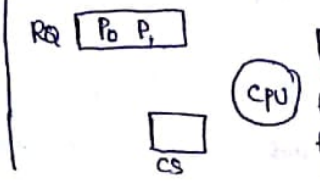
Mutual Exclusion : At any time only one process is present in CS.

Progress : If turn got to P_i and it is not intrested in Entering to CS but it stops the process P_j to Enter CS

Bounded waiting: After accessing CS we are making turn = j, this stops the same process Entering again and again in CS.

③ peterson solution (Toss of a cricket) (Lock var + strict Alternation)

→ Busy waiting
→ s/w solution
→ 2 process Mech



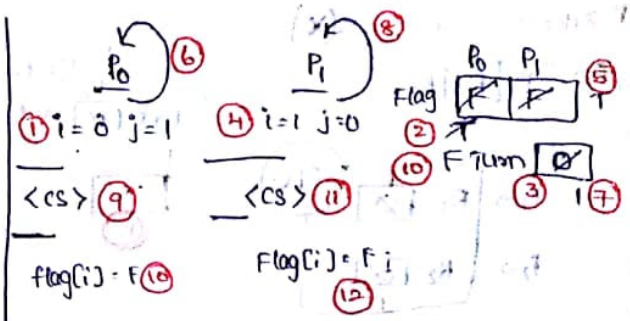
Result :

ME: ✓
Progress: ✓
Bounded waiting: ✓

☹
- Applicable to only 2 process
- It leads to wastage of CPU time due to busy waiting.

Implementation :

```
#define N 2
#define TRUE 1
#define FALSE 0
int flag[N] = {FALSE};
int turn;
void process (int i)
{
  int j = Not i;
  while(1)
  {
    a) Non_cs();
    b) flag[i] = TRUE
    c) turn = i;
    d) while ( flag[j] == TRUE && turn == i);
    e) <cs>
    f) flag[i] = FALSE
  }
}
```



The process which goes into busy wait first will access the cs first.

Mutual Exclusion :

t_1 : P_0 : a; b; pre.
t_2 : P_1 : a; b; c; d↑
t_3 : P_0 : c; d↑
t_4 : P_1 : d; e(cs)

ME Guranteed

Progress : The process running in Non_cs() doesn't affect the intersted process if its flag value is F

Bounded waiting : It gurantees Bounded waiting because after accessing the CS we are making flag[i] = F & turn = i again so that the other process will get Chance to Enter CS.

(4) TSL (Test and Set Lock)

→ Busy waiting
→ H/w solution
→ Multiprocess sol

→ Extension to lock var mech
∵ overcomes premption

Result :
ME : ✓
Progress : ✓
Bounded waiting : ×

*) With some modifications we can make TSL gurantees Bounded waiting

Syntax : TSL(Target) : returns the current value of lock (pointed by target) & set the value of lock (through target) to always 'TRUE'

read, write → Atomically Executed without premption.

Implementation :

```
Bool lock = FALSE
Void process (int i)
{
  while(1)
  {
    a) NON_CS();
Entry  b) while ( TSL (&lock) == TRUE) ;
    c) <CS>
Exit   d) lock = FALSE;
  }
}
```

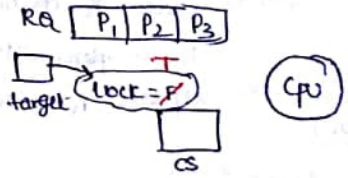
```
Bool TSL (Bool *target)
{
  Bool rv;
  a) rv = *target
  b) *target = TRUE
  c) return rv
}
```

Previlged Instructions ie Executed without Premption

Mutual Exclusion :

t_1 : P_1 : a; b; pre (TSL→F)
t_2 : P_2 : a; b (TSL→T)
t_3 : P_1 : b; c; pre
t_4 : P_2 : b (TSL→T)
t_5 : P_1 ; d ;
t_6 : P_2 : b; c; d .

ME ✓



Progress : The process running outside cs doesn't affect the onbested process

Bounded waiting : The process that made lock = False after accening cs will again go into CS by making TSL → F.

(5) SWAP :

→ Busy waiting
→ H/w solution
→ Multiprocess solution

→ extension to lock var mech
∵ overcomes premption

Result :
ME : ✓
Progress : ✓
Bounded waiting : ×

*) with some modifications in the logic (by Introducing Round Robin fashion) we can make swap gurantees Bounded waiting.

Implementation :

```
Bool lock = FALSE
Void process (int i)
{
  Bool key ;
  while(1)
  {
    a) Non_CS() ;
    b) key = TRUE ;
    c) do {
         swap (&lock, &key);
       } while (key == TRUE);
    d) <CS>
    e) lock = FALSE ;
  }
}
```

Entry

Exit

```
Void swap (Bool *a, Bool *b)
{
  Bool temp ;
  temp = *a
  *a = *b
  *b = temp ;
}
```

Previlged Ins Executed w/o Premption.

Mutual Exclusion :

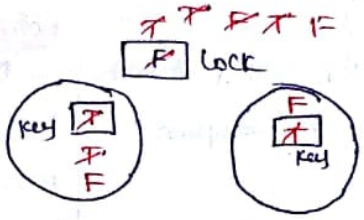
t_1 : (P1) : a, b ; pre

t_2 : (P2) ; a, b, c(swap) ; pre .

t_3 : (H) : c

t_4 : (P3) : c ; d ; e

t_5 : (P1) : c ; d ; e



ME ✓

Progress : The other process running outside cs doesn't affect interested process

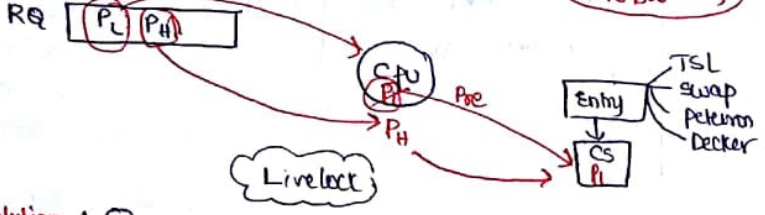
Bounded waiting : The process after accesing cs again enters into cs by swapping and making its key values as False.

*| Does TSL | swap gurantees Bounded waiting | (without mentioning code) : Yes

Priority Inversion problem : (Arise only in Busy waiting Sync Mech)

Preemptive priority based scheduling :

So we prefer NBW sols



Deadlock	Livelock
State : Blocked	State : Ready Running
∞ waiting	
finite waiting — spinlock.	

Solution : ① Priority Inheritance : Desirable

make P_L as high priority than P_H

P_L Completes its Execution

Update P_L with its own priority

② Round Robin Scheduling

Not desirable bcoz P_L gets equal chance as P_H.

⑥ Sleep() and wakeup() :

→ Non Busy waiting
→ OS based sol
→ Multiprocess sol

Implementation : (Producer Consumer problem)

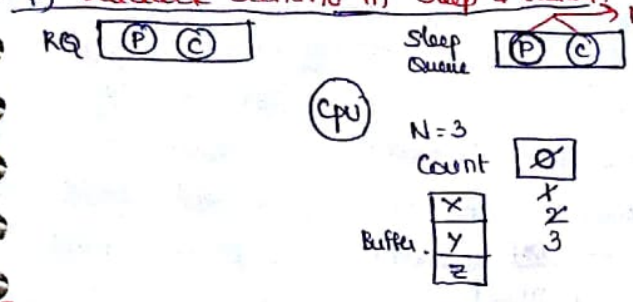
```
int count = 0;

Void producer (void)
{
  int itemp, in = 0;
  while(1)
  {
    a) itemp = produceitem();
    b) if (count == N)
         sleep();
    c) Buffer[in] = itemp
    d) in = in+1 mod N
    e) count = count + 1
    f) If (count == 1)
         wakeup(Consumer);
  }
}

Void consumer (void)
{
  int itemc, out = 0;
  while(1)
  {
    a) If (count == 0)        → load, cmp, jump
         sleep();
    b) itemc = Buffer[out]
    c) out = out + 1 mod N
    d) count = count - 1
    e) process item(itemc);
    f) If (count == N-1)
         wakeup(producer);
  }
}
```

Process Cooperation Deadlock.

Process competition Inconsistency Dataloss

Scanned by CamScanner

*) Deadlock scenario in Sleep & wakeup.

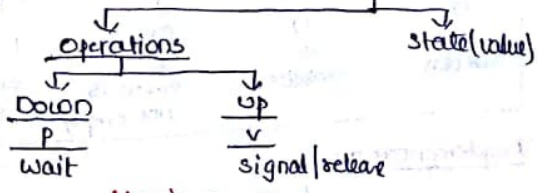


⑦ Semaphores :

- → Non Busy waiting
- → OS based sol
- → ~~Separate~~ Multiprocess mech

- → General purpose utility
- → Extension to sleep & wakeup
- → Developed by Dijkstra

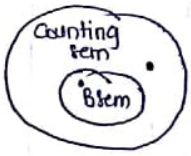
*) Implemented as Abstract data type



Atomic operations

Def of semaphores :

Semaphore is a variable of semaphore datatype that takes only integral values.



Based on range

Counting semaphore	Binary semaphore
General	Mutex
$<-\infty$ to $\infty>$	$<0,1>$

Counting Semaphore :

Def :
```
struct
{
  int value ;
  QueueType L ;  => List of all PCB's of the process that get blocked while
} CSEM ;            performing Down operation unsuccesfully.
```

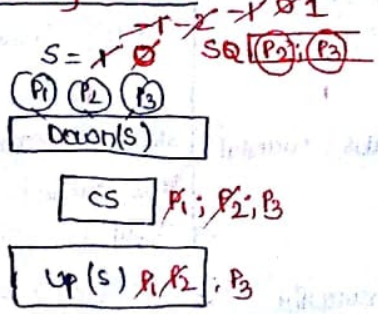
Implementation :

```
CSEM ;
S.value = 4 ;
:
Down(S)
<Si>
```

```
Down(S)
{
a) S.value = S.value - 1
b) If (S.value < 0)
   {
     Put that process (PCB)
     in S.L (Queue) &
     Block it (sleep implicitly)
   }
   Else
     return ;
}
```

```
up (CSEM S)
{
 a) S.value = S.value + 1
 b) If (S.value <= 0)
    {
      select a process from S.L (Queue)
      wakeup()
    }
    Else
      return ;
```

Concluding remarks :



*) If S.value =

4	Down
3	Down
2	Down
1	Down
0	Down
-1	Blocked
-2	Blocked
-3	Blocked

Succesfull Downs = 4

Blocked process = 3

SQ process = 3

Sleeping Queue

FIFO	LIFO, priority
No starvation	starvation
→ Strong semaphores	→ weak semaphores

*) S = 8 ; 10p ; 1V ; 18p : 6V ; 3p ; 4V ; 5p : 1V S = ? -16

*) S = 13 ; 15p ; 4V : 2p : 3V ; 9p ; 1V S = -5

x - 15 + 4 - 2 + 3 - 9 + 1 = -5

x - 18 = -5 ⇒ x = 13

Binary semaphore : (Mutex)

1 Down 0 ↓ succes	0 Down 0 ↓ unsucces	0 up 0 ↓ Queue is not Empty	0 up 1 Queue is Empty	1 up 1 Queue is Empty

Def :

```
struct
{
 enum value (0,1)
 QueueType L;
} BSEM;
```

L ⇒ List of all PCB's that are blocked while performing down unsuccesfully.

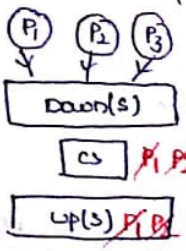
Implementation :

```
BSEM s
s.value = 1
;
Down(s)
<Si>
```

```
Down (BSEM S)
{
 s.value = s.value - 1;
 If (S.value == 0)
 {
  Put this process (PCB)
  in S.L (Queue) &
  Block it (sleep Implicitly)
 }
 Else
  S.value = 0
  return;
}
```

```
up (BSEM S)
{
 If (S.L(Q) is not Empty)
 {
  select a process from S.L (Queue)
  and wakeup();
 }
 Else
  S.value = 1;
}
```

Ex:-



S = 1 0 0 0 0 0

SQ P2 P3

CS P1 P2 P3

up(s) P1 P3

*) If S=0 ⇒ we can't say number of blocked Process
⇒ we may have blocked or unblocked Process
⇒ we can conclude that a prosem is present in cs.

*) S = 1 ; 10p ; 2V ; 3p ; 1V ; 18p ; 3V

SQ (9P) (7P) (8P) (9P) (27P) (24P)

S=0 , SQ Contains 24 proces.

Cases of Binary semaphore :

① S = 1
P(s)
S = 0
Status = succesfull

② S = 0
P(s)
S = 0
Status = unsuccesful

③ S = 0
V(s)
S = 0 ⇒ Queue not Empty
1 ⇒ Queue Empty
Status = Succesful

④ S = 1
V(s)
S = 1
Status = Succesful

⑤ S=0 (Initial)
V(s) (First open)
S = 1
Status = succesfull
but Queue is definitely Empty

*) The status of Down operation may be succesful or unsuccesful.

*) The status of up operation is always succesful.

Classical IPC problems :

Producer Consumer Implementation using semaphore :

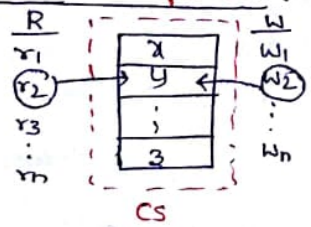
```
#define N 100
CSEM Empty = N;   <Num of Empty slots in buffer>
CSEM full  = 0;   <Num of full slots in buffer>
BSEM Mutex = 1;   <Used by (P) & (C) to access buffer as CS>
```

```
void producer(void)
{
  int itemp, in = 0;
  while(1)
  {
    a) Non_CS()
    b) itemp = produce item();
    c) Down(Empty);
    d) Down(Mutex);
    e) Buffer[in] = itemp;
    f) in = (in+1) mod N
    g) up(Mutex);
    h) up(full);
  }
```

```
void consumer(void)
{
  int itemc, out = 0
  while(1)
  {
    a) Non_CS();
    b) Down(Full);
    c) Down(Mutex);
    d) itemc = Buffer[out];
    e) out = (out+1) mod N;
    f) up(Mutex);
    g) up(Empty);
    h) process_item(itemc);
  }
```

Entry, CS, Exit, Co-operation

Readers writers problem :



① First Reader writer :

Priortize reader if reader already present in CS.

Case 1 : r (or) w : Trivial (Allowed)

Case 2 : r and w arrived simultaneously (Anybody allowed)

If reader allowed first

t_1 : r_1 ✓ W_1

t_2 : r_2 ✓

t_3 : r_3 ✓

t_4 : W_2

t_5 : r_4 ✓

→ writer starvation

→ writer gets chance if all readers are completed

If writer allowed first

t_1 : W_1 ✓ r_1

t_2 : r_2

t_3 : W_2

t_4 : W_3

t_5 : r_3

→ After W_1 any one can access CS.

Implementation of First reader solution using semaphore :

```
int rc = 0 ; (Reader count)
BSEM Mutex = 1 ;   <used by (R's) to acen rc as CS>
BSEM db = 1 ;   < used by (Rs) and (Ws) to accen DB as CS>
```

```
void writer()
{
  while(1)
  {
    a) Down(DB)   Entry
    b) <write to DB> CS
    c) up(DB)   Exit
  }
}
```

```
Void reader()
{
  while (1)
  {
    a) Down(Mutex)   Entry
    b) rc++;
    c) if (rc == 1)   CS
         Down(DB)
    d) up(Mutex) ;   Exit   Entry
    e) <Read the DB>   CS
    f) Down(Mutex)   Entry
    g) rc--;
    h) if (rc == 0)   CS
         up(DB);
    i) up(Mutex);   Exit   Exit
```

*) 1st reader is the leader of all other readers

1st reader → Accen DB : All other R's Accen DB
1st reader → Blocks : All R's are blocked.

First writer reader problem / second reader writer problem :

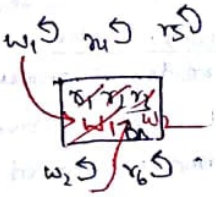
Prioritize writer

Case 1 : (R) / (W) : Trivial (Allowed)

Case 2 : (R) and (W) arrived simultaneously (writer Allowed)

Special case :

$t_1 \rightarrow r_1$
$t_2 \rightarrow r_2$
$t_3 \rightarrow r_3$
$t_4 \rightarrow w_1$
$t_5 \rightarrow r_4$
$t_6 \rightarrow r_5$
$t_7 \rightarrow r_1 r_2 r_3$ ×
$t_8 \rightarrow w_1$ (DB) priority @ writer
$t_9 \rightarrow w_2$
$t_{10} \rightarrow r_6$
$t_{11} \rightarrow w_1$ ×
$t_{12} \rightarrow w_2$ ✓ prioritize writer



Solution :

```
int rc=0, wc=0 ;
Bsem read=1, rmutex = 1, wmutex = 1, DB=1;
void reader()
{
  a) Down(read)
  b) Down (rmutex)
  c) rc++
  d) if(rc==1) Down(DB);
  e) up(rmutex);
  f) <CS>
  g) Down(rmutex)
  h) rc--
  i) if(rc==0) up(DB);
  j) up(rmutex)
}
```

```
Void writer()
{
  a) Down(wmutex)
  b) wc++;
  c) if (wc==1) Down(read);
  d) up(wmutex)
  e) Down(DB)
  f) <CS>
  g) up(DB);
  h) Down(wmutex) ;
  i) wc--
  j) if (wc==0) up(read);
  k) up(wmutex);
}
```

Dining philosophers problem :

$N \geq 2$

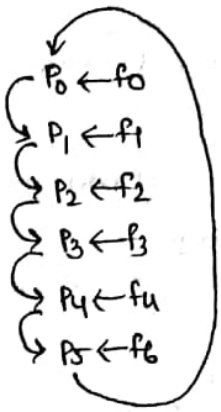
Max Concurrency = 3.

Non semaphore based soln :

1. $(N-1)$ p $\Rightarrow$ L-R
   1 p $\Rightarrow$ R-L
2. odd p $\Rightarrow$ L-R
   even p $\Rightarrow$ R-L
3. Providing one more Extra fork.

Implementation of Dining philosophers problem :

```
#define N 6
void philosopher (int i)
{
  while(1)
  {
    a) Think(i);
or  b) take_fork(i);
    c) take_fork((i+1)%N);
    d) Eat(i)
    e) put_fork(i);
    f) put_fork((i+1)%N);
  }
}
```



deadlock.

solution → semaphore based soln

Semaphore based solution :

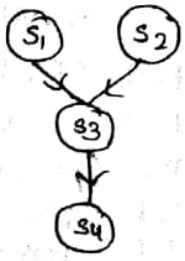
```
#define N 6 ; BSEM S[N] = {1};
void philosopher (int i)
{
  while (1)
  {
    a) Think(i);
    b) Down ( take_fork (S[i]));
    c) Down ( take_fork (S[i+1] mod N);
    d) Eat (i);
    e) up (put_fork(S[i]));
    f) Up ( Put_fork (S[i+1] Mod n);
  }
}
```

Concurrent programming :

Concurrency mechanisms :

```
→ {
    S1 : a = b + c
    S2 : d = e * f
    S3 : k = a + d
    S4 : l = k * m
```

Precedence graph



Concurrency types :

- Real | physical (Multi cpu | multicore)
- Interleaved | pseudo (uniprocessor system by Context switching)

Concurrency conditions :

S ← $R(S)$: Readset, $W(S)$: Writeset

Ex:- $S: a = ++b * ++c$.

$R(S) = a, b, c$

$W(S) = a, b, c$

Bernsteens Concurrency conditions

S_i < $R(S_i)$, $W(S_i)$; S_j < $R(S_j)$, $W(S_j)$

1. $R(S_i) \cap W(S_j) = \phi$
2. $R(S_j) \cap W(S_i) = \phi$
3. $W(S_i) \cap W(S_j) = \phi$
4. $R(S_i) \cap R(S_j)$ = may | may not ϕ

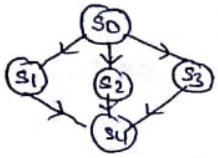
Concurrency Mechanisms : ① (parbegin – parend) | (cobegin – coend)

② Fork and Join

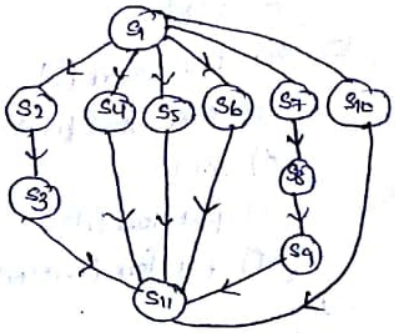
(Parbegin – parend | Co-begin – Coend) :

*)
```
{
  S1
  S2
  S3
}
```

*)
```
S0;
Par begin
  S1;
  S2;
  S3;
Par End
S4;
```

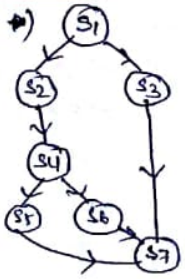


* (Model 1)
```
S1
Par begin
  begin S2; S3; End
  S4
  S5
  S6
  begin S7; S8; S9; End
  S10
Par End
S11;
```



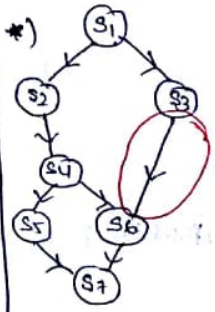
Model 2 :

*)



→
```
S1
parbegin
  begin S2; S4
    par begin
      S5
      S6
    Par End
  End
  S3
ParEnd
S7
```

*)



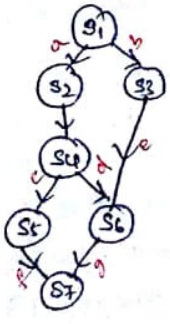
⇒ Non-Implementable

But Implementable using Semaphores.

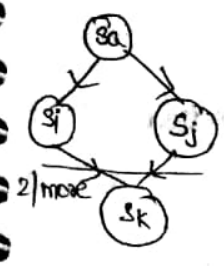
Par begin – parend with Semaphore :

↳ All process are blocked

```
BSem a, b, c, d, e, f, g = 0/1 ; Initially
Par begin
  begin S1 ; V(a) ; V(b) ; End
  begin P(a) ; S2 ; S4 ; V(c) ; V(d) ; End
  begin P(b) ; S3 ; V(e) ; End
  begin P(c) ; S5 ; V(f) ; End
  begin P(d) ; P(e) ; S6 ; V(g) ; End
  begin P(f) ; P(g) ; S7 ; End
```



Fork and Join :



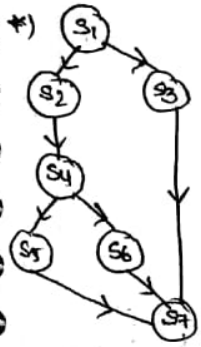
Syntax of fork :

```
Sa , Count = 2 ;
fork L
Si
:
goto L1
L : Sj
L1 : join(count) ;
Sk ;
```

Syntax of join : Join(Count).

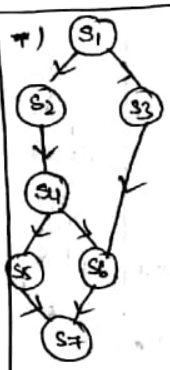
```
Join (int Count)
{
  Count = Count - 1;
  if (count ≠ 0)
  Exit
  Else
    return ;
}
```

*)




```
S1 , Count = 3
Fork L1
S2;
S4 ;
Fork L2
S5 ;
goto L3 ;
L1 : S3  goto L3
L2 : S6  goto L3   redundant
L3 : join (Count)
S7
```

*)

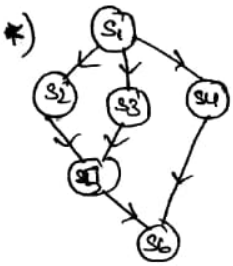



```
S1 , a = 2 , b = 2
Fork L1
S2
S4
Fork L2
S5
goto L3
L1 : S3  goto L4
```

~~S2 ; S6 goto L3~~
~~L2 : join(a)~~
~~L3 : join(b)~~

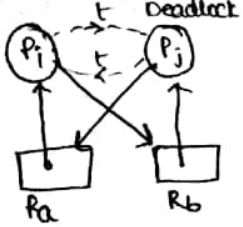
```
L4 : join(b)
L2 : S6  goto L3
L3 ; join(a)
S7
```

*)




```
S1 ; a = 2 ; b = 2 .
Fork L1
S2 ;
goto L2
L1 : Fork L3
S3
L2 : join(a)
S5
goto L4
L3 ; S4  goto L4
L4 : join(b)
S6
```

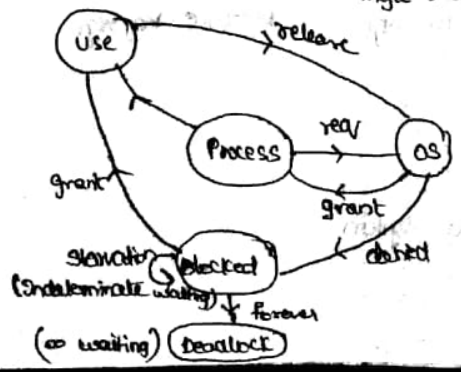
Deadlocks :



System model :

Consider n processes m resources

- Single Instance
- multiple Instance

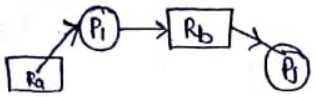


Characterization :

Necessary Conditions :

a) **Mutual Exclusion**: CS (shared resource)

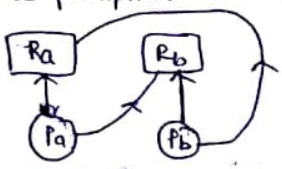
b) **Hold & wait** :



⇒ Every hold & wait doesn't imply deadlock.

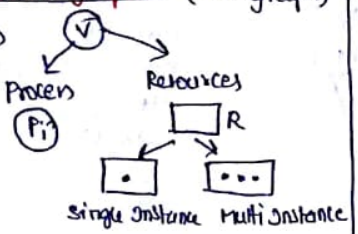
c) **No premption** : No premption of resources may cause deadlocks

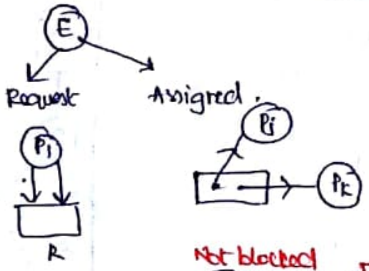
d) **Circular wait** :



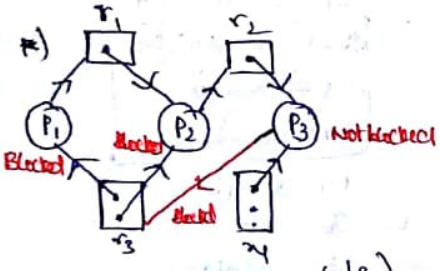
Resource Allocation graph : (Multigraph)

$G(V, E)$ ⇒





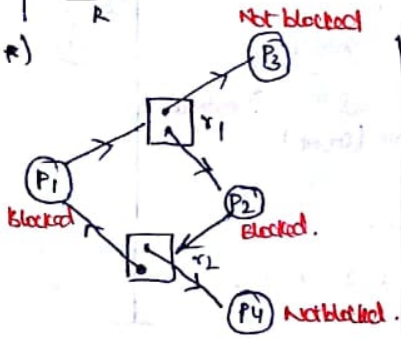
*)



$P_1 - r_1 - P_2 - r_2 - P_3 - r_3 - (P_1/P_2)$ ⇒ cycle.

Deadlock.

*)



$P_1 - r_1 - P_2 - r_2 : P_1$ ⇒ cycle.

No Deadlock.

*) Deadlock may occur if P_3 req for r_2 and P_4 req r_1

*) Every Dead lock Contains cycle but Every cycle doesn't lead to deadlock

Strategies

After ↓

Deadlock happend

- Deadlock Detection & Recovery (Doctors' Algo)
- Deadlock Ignorance (Ostrich Algo)

Before Deadlock happened

- Deadlock prevention
- Deadlock Avoidance (Banker's Algo)

Deadlock Ignorance :

Ostrich Algo

↓

Restart the Computer. → Djikstra's

Deadlock Avoidance : (Banker Algo)

- Safety Algo
- Resource req Algo

System State :

safe
unsafe

Datastructures that govern state of a system

(1) n : num of process

③ Max[1--n, 1--m]$_{n\times m}$ ⇒ Max[i,j] = k ie $P_i \xrightarrow[Max]{req}$ K copies of R_j

④ Alloc [1--n, 1--m]$_{n\times m}$ → Alloc[i,j] = b ie $P_i \xleftarrow[Alloc]{}$ b copies of R_j

⑤ Need [1--n, 1--m]$_{n\times m}$ → Need[i,j] = c ie $P_i \xrightarrow[Need]{}$ c copies of R_j

⑥ Request [1--n, 1--m]$_{n\times m}$ ⇒ Req$_t$[i,j] = d @ t ie $P_i \xrightarrow[req_t]{}$ d copies of R_j @ t.

⑦ Total [1--m] ⇒ Total[j] = 'z' ie There are 'z' copies of R_j

⑧ Avail[1--m] ⇒ Avail[j] = 'y' ie There are 'y' copies of R_j available @ t

Safety Algo : (Single Instance)

	Max R	Alloc R	Need R	Avail R
P_1	5	3	2	~~3~~
P_2	10	5	5	~~6~~
P_3	15	7	8	~~11~~
P_4	8	4	4	~~18~~
P_5	6	3	3	~~21~~
		22		25
		Safe		Total

order : $P_1 - P_2 - P_3 - P_4 - P_5$

Multiple Instance :

Total → 10 5 7

	Max ABC	Alloc ABC	Need ABC	Avail ABC
P_0	753	010	743	332
P_1	322	200	122	532
P_2	902	302	600	743
P_3	222	211	011	745
P_4	433	002	431	755
		725		<10,5,7> Total

Safe

order : P_1 ; P_3 ; P_4 ; P_0 ; P_2

Resource request Algo :

```
Algo Resource_req (Pi, reqi, Alloci, Needi, Avail)
{
  a) Reqi <= Needi
  b) Reqi <= Availi
  c) [Assume to have satisfied Reqi]
       (i)  Availi = Aval - Reqi
       (ii) Alloci = Alloci + Reqi
       (iii) Needi = Needi - reqi
  d) Run safety Algo
  e) If (system == safe)
       Grant resources
  f) Else
       Deny the request
}
```

Consider this Ex.

Ex:- t_1: P_1 → <1027 req_1

	Max ABC	Alloc ABC	Need ABC	Avail ABC
P_0	753	010	743	~~332~~
P_1	322	~~200~~ 302	~~122~~ 020	<230>
P_2	902	302	600	
P_3	222	211	011	
P_4	433	002	431	

Run safety Algo
<532>
<743>
<745>, <755>
<10,5,7>

order : P_1 ; P_3 ; P_4 ; P_0 ; P_2

Safe → req granted. ∴ Avail = <230>

t_2: P_4 → <330> req_4 rejected bcoz. $Req_i > Avail$

t_3: P_0 → <020> req_0

	Max ABC	Alloc ABC	Need ABC	Avail ABC
P_0	753	~~010~~ 030	~~743~~ 723	~~230~~
P_1	322	302	020	<210>
P_2	902	302	600	
P_3	222	211	011	
P_4	433	002	431	

order : × Unsafe → req denied.

(3) Deadlock detection & Recovery :

*) Banker's Algo :- System → safe | unsafe.

Detection Algo :- safe | deadlock.

Detection Algorithm :

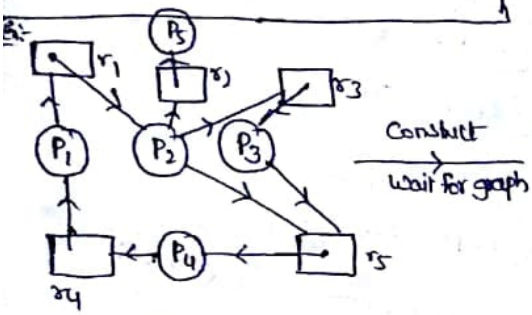
When to Apply ?

⇒ Throughput drastically decreases
⇒ Majority process in the main memory are blocked.

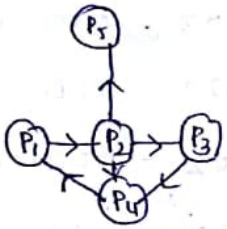
Detection when resources are single instance :

1. Construct wait for graph
2. Run cycle detection Algo
3. if cycle is present then
   Deadlock
   Else
   No deadlock.

*) In single instance resource cycle is necessary and sufficient to say a process is in deadlock.



Construct wait for graph



Run cycle detection algo

P1 – P2 – P3 – P4 – P1 (deadlock)

*) In multi instance resource cycle is only necessary condition for happening of deadlock.

Detection when resources are multi instance :

n=5, m=3 ; <ABC> = <7 2 6> Total.

to	Alloc ABC	Req ABC	Avail ABC
P0	010	000	~~000~~ 7 2 6
P1	200	202	~~010~~ 5 1 3
P2	303	000	~~3 1 3~~ 3 0 3
P3	211	100	5 1 3
P4	002	002	7 2 6

P0, P2 → not blocked
P1, P3, P4 → Blocked
↓ majority
Run detection Algo

At to ; P0 ; P2 ; P1 ; P3 ; P4 ⇒ safe

At t1 : P2 req → <0 0 1>

t1	Alloc ABC	Req ABC	Avail ABC
P0	010	000	~~000~~
P1	200	202	010
P2	303	001	
P3	211	100	
P4	002	002	

At t1 : P0 ; ⇒ Deadlock

Blocked process : P1 ; P2 ; P3 ; P4

Deadlock recovery :

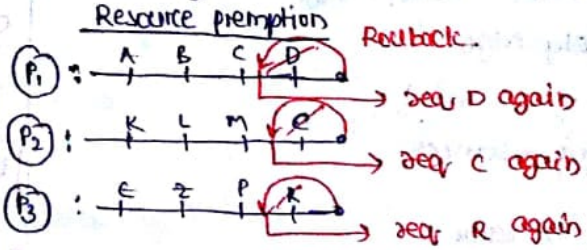
Deadlock Process → i/p → Deadlock Recovery → o/p → Deadlock free Process.

Deadlock Recovery :

- Process Termination
  - KILL ALL
    → No need to apply Detection algo again
    ∴ Restart all process
  - KILL ONE AT A TIME
    ∴ Need to apply detection algo repeatedly.
- Resource premption (Both suffer from starvation)

*) All killed process need to restart from the beginning

*) Process Termination and Resource premption suffer from starvation.



Rollback.

→ prempt latest resource of all process
→ Run detection Algo

Deadlock prevention :

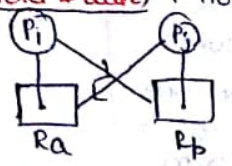
Dissatisfy one | more necessary conditions of the deadlock

→ Mutual exclusion

1) Mutual Exclusion : (X)

Mutual Exclusion is non-dissatisfiable due to the fact that system contains atleast one shared resource.

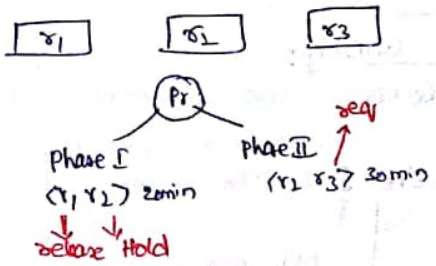
2) (Hold & wait) : Hold or wait



Protocol 1 : process must request and be allocated to all the resources prior to its commencement

∴ under utilization of resources

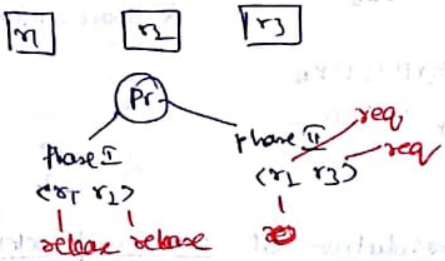
Starvation.



Holding or waiting happened

Protocol 2 : Release all the resources before making a fresh or new request

∴ starvation may happen



only waiting happens

3) ∴ (No premption) → premption of resources.

Forceful	self
→ selfish spirit	→ self less spirit
→ process running on cpu forces other to prempt the resources	→ process running on cpu prempts all its resources and goes to block state.

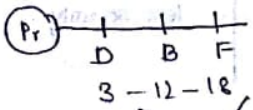
4) Circular wait :

Circular wait is dissatisfied by (i) Number all resources uniquely

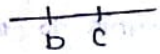
(ii) Never allows a process to req a lower num resource than the last one req & allocated.

Ex:-

Res	RDS_Del
A	9
B	12
C	8
D	3
E	25
F	18
G	28



prempt F & B.

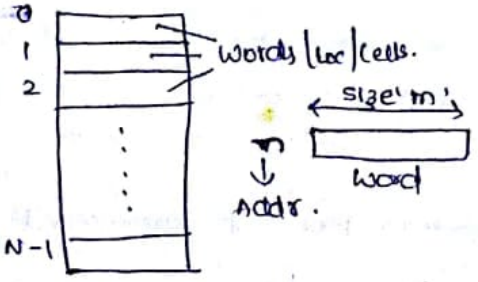


req for F & B → may | may not Available

∴ Suffer from Starvation.

Memory Management :

Abstract view of memory : (1D Array of words)



N → num of words
n → Address.
m → word size (bits | byte)

$N = 2^n$ words
$n = \log_2 N$ bits.
$N \propto n \propto m$

loading vs linking :

Prog size = 60 KB.

Loading :

```
main( ) 5KB          g( ) 25KB
{                    {
  :                    :
  if(cond)             h(); BSA __
BSA __ f();          }
  :                  h( ) 8KB
}                    {
f( ) 10KB              :
{                    sqrt(); 12KB
  :                    :  BSA __
BSA __ g()           }
}
```

loading
- Static
  - ☺ faster Execution
  - ☹ space inefficient
- Dynamic (loading on demand basis)
  - ☺ space efficient
  - ☹ Time inefficient

Linking : It is resolution of unresolved references

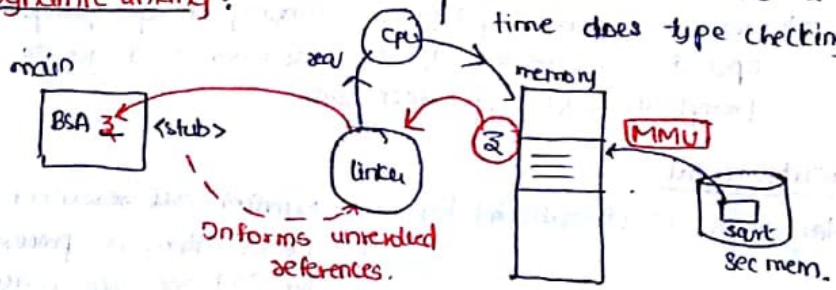
- fn's
  - Library
  - Userdefind.
- var's
- objects.

Resolution :

Static linking :

static linking
- ☺ Time Efficient, More secured.
- ☹ space inefficient

Dynamic linking :



- ☺ Space Efficient, Reusability, flexibility
- ☹ Time inefficient, less security.

Imp

*) Header files contains prototype of a fn :
Prototype → ret type, Name, Args profile.

*) Definition of fn is present in library with (.lib) Extension)

*) Declaration of fns at compile time does type checking

Address binding :

Association of program instructions and data units to memory locations (having addresses)

Deciding where to load — Address binding
Loading of ins & data — loader
} difference.

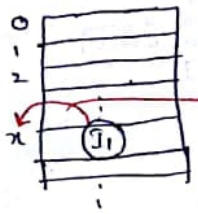
Binding
- static binding : Compile time, load time
- dynamic binding : Runtime.

```
int a,b,c;
a = b+c;
```

Compiler

4000	a	int	2
4002	b	int	2
4004	c	int	2

I_1 : L, R_1, b 2000
I_2 : L, R_2, C 2002
I_3 : A, R_1, R_2 2004
I_4 : S, a, R_1 2006.



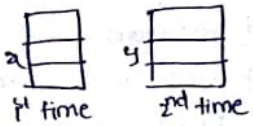
Decision by Address binding

Binding time

Compile time
- Deciding the loc of ins/data where to load
- Allocation is not happened at Compile time
- static binding
  Ex:- DOS
- ∵ Inflexibility
  More burden to Compiler

Load time
- If Compiler doesn't do Addr binding then loader will do Addr binding
- Compiler generates logic Addresses like
  I_1 : 0
  I_2 : 2
  I_3 : 4
  Logical addr
- loader at loadtime in memory with base addr 2000 loads
  I_1 : 2000+0 : 2000
  I_2 : 2000+2 : 2002
  I_3 : 2000+4 : 2004
  physical addr
- static binding

Runtime.
- Dynamic binding
- ∵ More flexibility due to dynamic relocation



Compile time binding:
If Compiler associates pgm instructions and data units to absolute address of memory

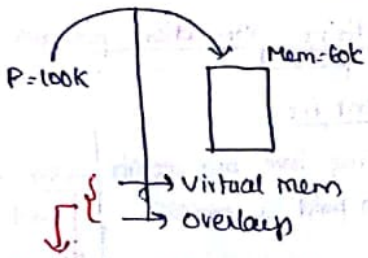
load time binding:
Compiler generates only logical offsets which are relocated to physical address by loader during load time

Runtime binding:
During Runtime binding pgms during Execution can get relocated in different areas of memory.

Functions & Goals of memory management:

Functions: (F PAD)
- Allocation
- protection
- Free space mgmt
- Deallocation.

Goals:
- effective utilization of memory space. (Minimizing fragementation levels)
- Managing Execution of larger pgms in Smaller mem area



Increases multiprogramming
↓
Throughput ↑

Memory management techniques:

Contigues
↓
centralized Allocation
Ex:- partitions (2)
overlays (1)
Buddy systems

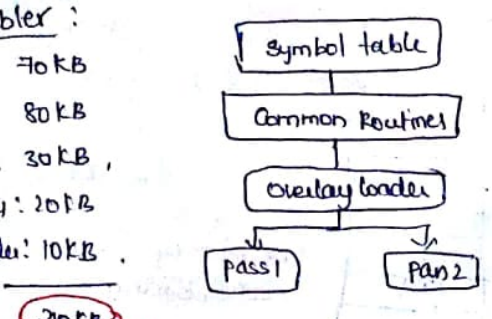
Non Contigues
- decentralized / distributed Allocation

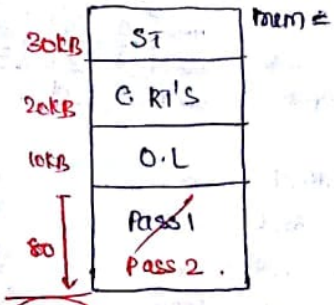
Ex:- paging
segmentation
segmented paging
virtual memory.

(1) Overlays:
- Contigues mem mgmt technique.

2 pass Assembler :

Pass 1 : 70 KB
Pass 2 : 80 KB
Symtable : 30 KB ,
Common Routin : 20 KB
Overlay loader : 10 KB .





→ Min memory reqd.

larger pgm to smaller mem.

Note :

*) In overlay trees the min memory required is the maximum path length.

② partitions :

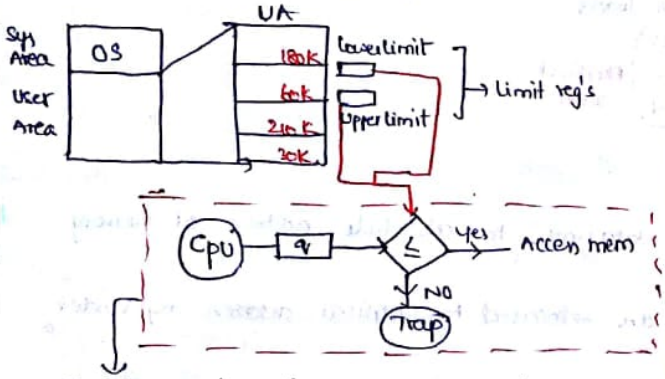
Fixed partitions

(multipgming with fixed tasks)

Rule : 1 partition = 1 pgm

→ Size is variable

→ Num of partitions are fixed.



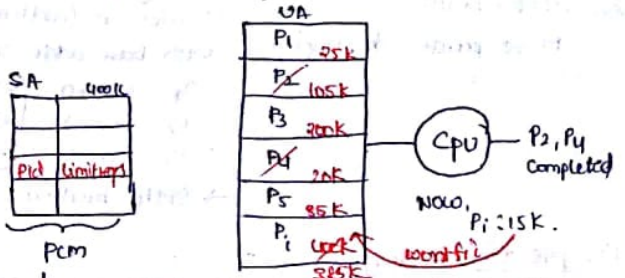
Provide security of non interfering of other pgms

→ static scheme wrt partitions.

Variable partition

(multipgming with variable tasks)

JP	P_1	P_2	P_3	P_4	P_5	P_6
req :	25K ;	105K ;	200K ;	20K ;	85K ;	310K.



→ It consists of info like partition id, Limit reg Attributes ...etc.

→ If two contigious free holes are available then automatically merge to one hole ⇒ Coalesces

→ Dynamic scheme wrt partitions.

Partition Allocation policies :

First fit

→ First free big enough to hold a process.

Best fit / Best Avail fit

→ Smallest free big enough to hold a process

- **of all** → Best fit
  ☹) waiting is present
- **of Available** → Best Avail fit
  ☺) No waiting
  • Default partion Alloc policy

Next fit

→ It works like first fit except that the search for free big Enough partition to hold a program Commences from the partition where the last alloc was made

☺ Works faster than first fit Num of search opers are less

Worst fit

→ Largest free big Enough to hold a program.

Performance Issues of fixed partition :

1) Internal fragmentation : ✓
2) External fragmentation : ✗
3) Degree of multipgming : limited to num of partitions
4) Max procen size runnable : limited to largest partition
5) Alloc policy : Best fit

Performance issue of variable partition :

1) Internal fragmentation : ✗
2) External fragmentation : ✓ → (It is considered when all the avail free holes are greater than the procen size)
3) Degree of multipgming : Flexible
4) Max procen size runnable : Flexible
5) Superior Alloc policy : Worst fit

So, only problem in variable partition

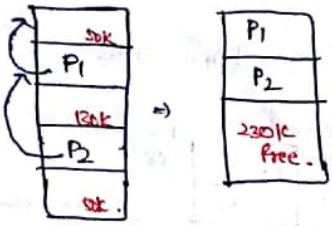
↓

External Fragmentation

↓ solution

↓ ↓

Compaction:



→ Compaction is supported by runtime binding

∴ Time Consumption process → bcoz of

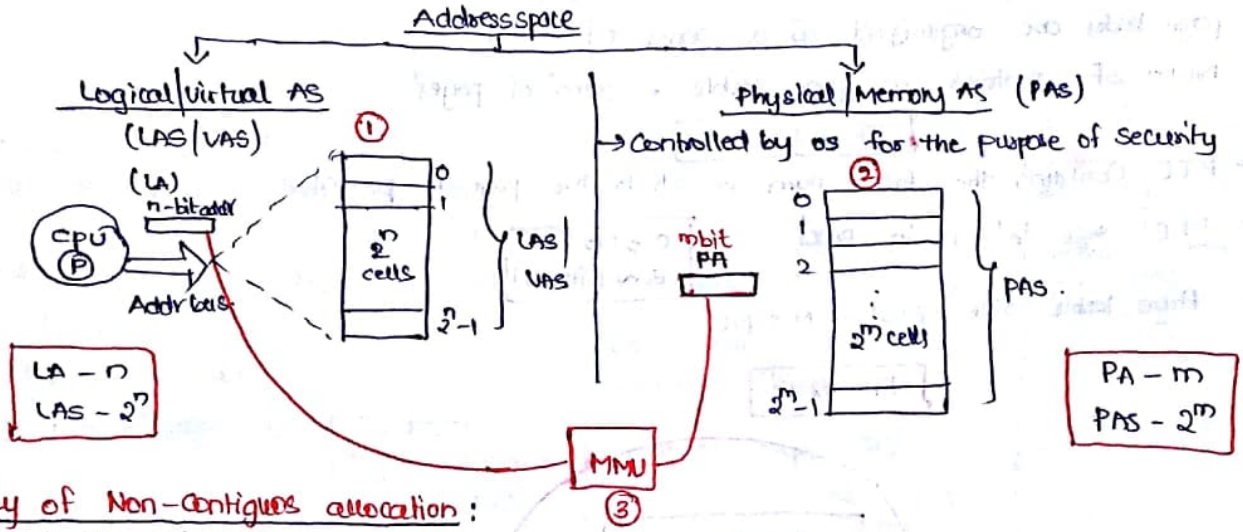
Non-Contiguous Allocation

→ paging
→ segmentation
→ segmented paging
→ virtual memory

Non-Contiguous memory Allocation:

Address space: (AS)

A Group of words or locations or memory cells associated with address is AS

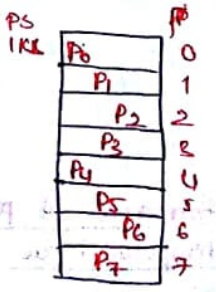


Study of Non-Contiguous allocation:

① organization of LAS
② organization of PAS
③ organization of MMU with Translation logic

PAGING:

LAS = 8KB PAS = 4KB Pagesize = 1KB
LA = 13 bits PA = 12 bits



① organization of LAS:

- The logical AS is divided into equal size units known as pages
- Generally page size is the power of 2
- Num of pages $(N) = \frac{LAS}{PS}$
- Each page is assigned with a unique num or id known as page number. given in bits

$$\text{Page No } (P) = \log_2 N = \log_2 \frac{LAS}{PS} \qquad N = 2^P = \frac{LAS}{PS}$$

- Page offset denoted by 'd', It is used to select a byte/word within a page.

$$d = \log_2 PS \qquad PS = 2^d$$

- Logical address format LA | P | d |

Points to remember :

LAS / PAS → Bytes PS → Bytes PTE → Bytes

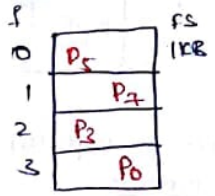
LA / PA → bits. P / d → bits.

② organization of PAS : (Page frame / Blocks / frames)

- PAS is divided into equal size units known as frames with the convention that

Frame size = page size

- Each frame holds 1 page of address space.
- Number of frames (M) = PAS/PS | PAS/FS.
- frame number (f) = $\log_2 M$ | $M = 2^f$
- frame offset = page offset
- physical address format PA | f | d |



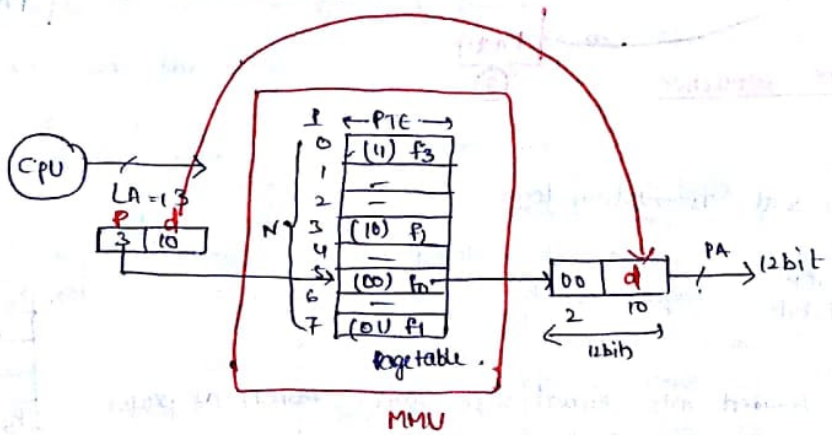
③ organization of MMU :

- Each process is associated with its own page table, which are stored in memory
- Page tables are organized as a series of Entries.
- Num of entries in page table = Num of pages

PT Entries = N

- PTE Contains the frame num in which the page is present
- PTE size 'e' is in Bytes. e ≥ 1B, e ∼ f (bytes)
- Page table size (PTS) = N * PTE (e)

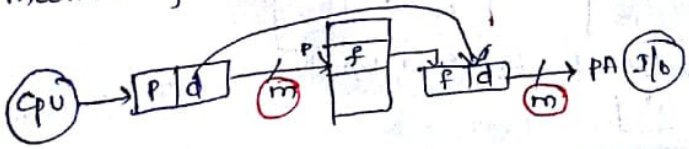
PTS = N * e



Performance of paging :

(1) Temporal issue : Goal : Reduce Effective mem access time (EMAT)
Improve throughput.

Let main memory access time be 'm'.

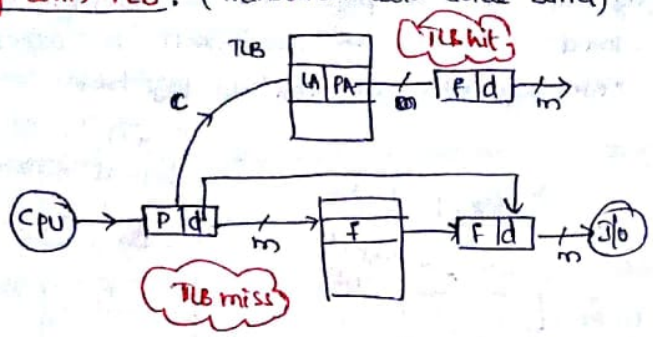


⇒ EMAT = 2m

⇓

To reduce EMAT from 2m to near m we using paging with TLB

Paging with TLB: (Translation look aside buffer)



C → TLB mem access time
x → hit ratio
1-x → miss ratio.

$$EMAT_{TLB} = x[C+m] + (1-x)[C+2m]$$

(1) Spacial issue: Goal: Reduce pagetable size.

$$PTS = N * e$$
$$PTS \propto N$$

Ex:- LA = 32 bits
PS = 4KB

$N = \frac{2^{32}}{2^{12}} = 2^{20} = 1M$ pages

Assume that each entry occupies 4B

∴ PTS = 4*1M = 4MB/process

For 100 processes

PTS = 400MB ⇒ Not desirable → Increasing page size / multilevel paging

sol

$PTS \propto N \propto \frac{LAS}{PS}$

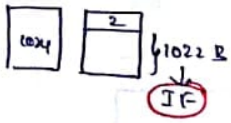
a) Increasing page size:

If PS = 8KB then N = 512k

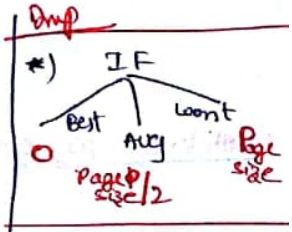
PTE = 512k entries.

But increasing page size will lead to fragmentation at last page.

Ex:- If ps = 1024 B
Prog size = 1026B
Num of pages = 2



Ps = 2B
Prog size = 1026B
Num of pages = 513
No IF



So, optimal page size should be selected.

sol
LAS - 's' Bytes
PTE - 'e' Bytes
PS - 'p' Bytes

$\Rightarrow PTS = \frac{s}{p} * e \qquad IF = \frac{p}{2}$

To reduce PTS & IF

$$\frac{d}{dp}\left(\frac{s}{p} * e + \frac{p}{2}\right) = 0.$$

$$\frac{-se}{p^2} + \frac{1}{2} = 0.$$

$$p = \sqrt{2se}$$

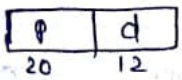
b) multi level paging: (Paging on page table)

Paging as a Concept Involves 3 steps

Step 1: Divide AS into Equal size pages.
Step 2: Store the pages in frames of PAS.
Step 3: Access the pages of the AS in the frames through a page table.

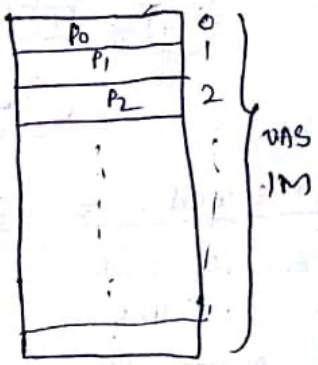
→ Multi level paging implies paging on page table ie the page table is divided into pages. The pages of the page table are stored in frames of PAS, which are accessed through another page table known as outer page table or first level page table.

Eg:- let
LA = 32 bit
PS = 4KB

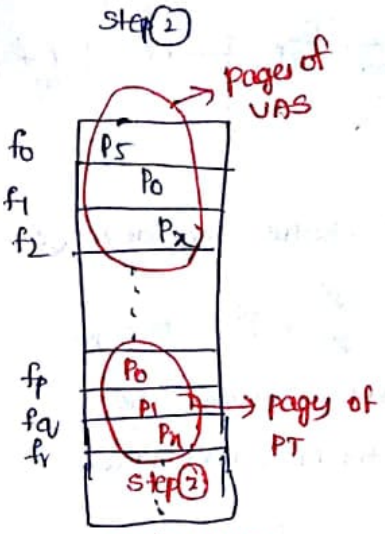


Let pages in PT = 1K
PS_{PT} = 1K

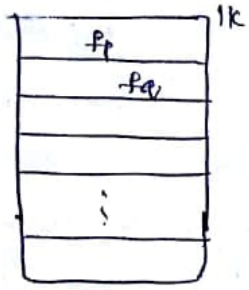
Step ① Logical Address space



Step ②

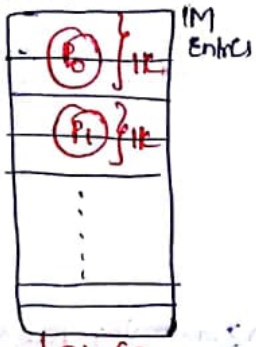


Outer PT
Step ③



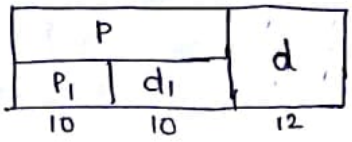
stored in mem

Inner PT Step ③

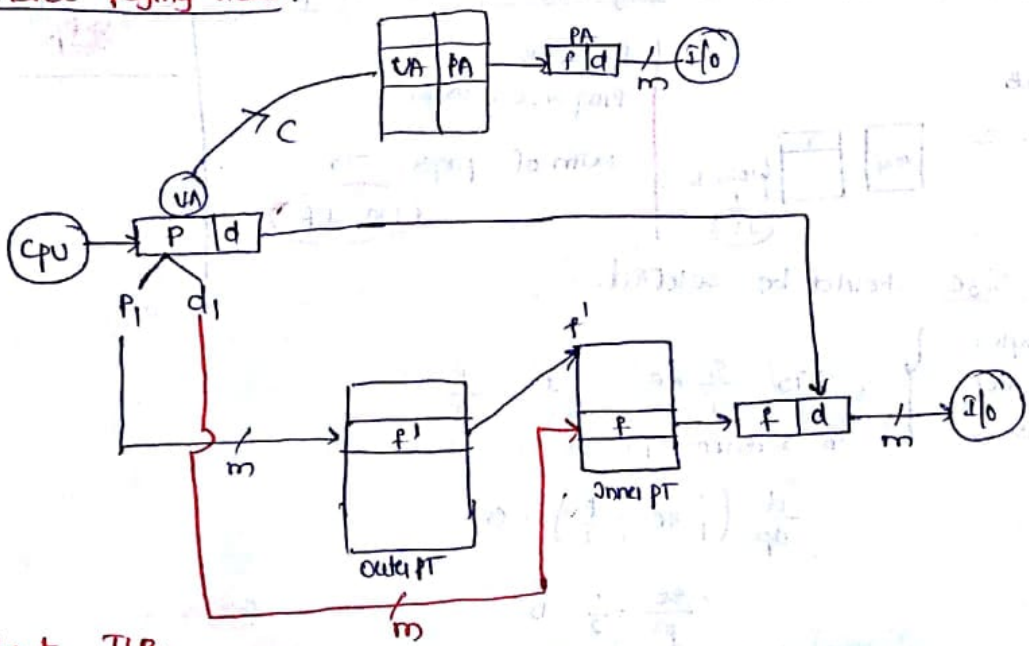


Step ① pushed to memory.

Address format :



Multilevel paging view :



Without TLB

$EMAT_{1lp} = 2m$

$EMAT_{2lp} = 3m$

⋮

$EMAT_{nlp} = (n+1)m$

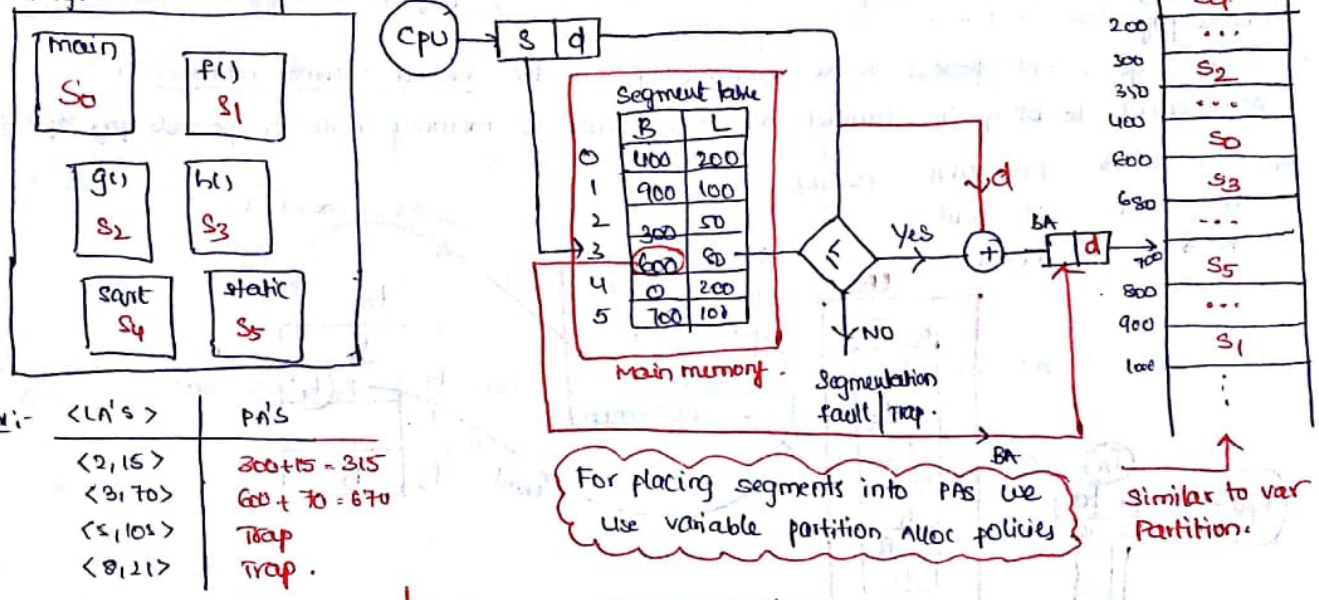
With TLB

$EMAT_{2lp\ TLB} = x[c+m] + (1-x)[c+3m]$

$EMAT_{nlp\ TLB} = x[c+m] + (1-x)[c+(n+1)m]$

Segmentation :

→ Paging doesn't preserve user view of memory allocation to programs.
→ As per user view of memory allocation program is divided into logical units known as segments.
→ A segment may represent a function, procedure, block or Datastructure
→ These segments are assumed to be stored in the main memory in their Entirety at non-contiguous location.
→ These segments are accesible through segment table by Addres translation
→ The segment table will have entries amounting to num of segments and each Entry contains the base addr of the segment in PAS and size of the segment.



For placing segments into PAS we use variable partition Alloc policies

Ex:-

<LA's>	PA's
<2, 15>	300+15 = 315
<3, 70>	600 + 70 = 670
<5, 101>	Trap
<8, 21>	Trap.

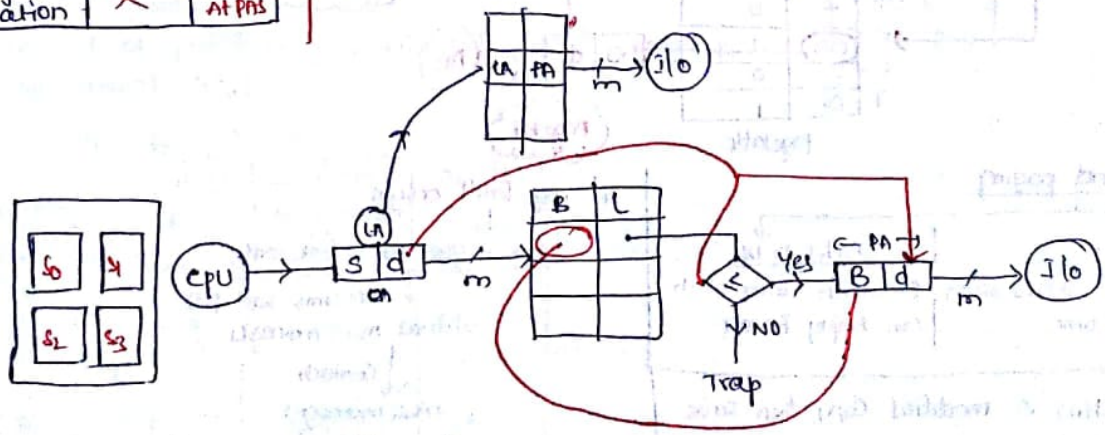
Paging vs segmentation :

	IF	EF
Paging	✓ Inpage	X
Segmentation	X	✓ At PAS

Performance of segmentation :

$$EMAT_{TLB} = \alpha[c+m] + (1-\alpha)[c+2m]$$

*)



*) paging is introduced in segmentation for 2 reasons
(i) To associate a smaller Address table with the process by applying paging on segment Table.
(ii) To overcome External fragmentation arising in PAS while storing segments.
In this case paging is applied on segments.
The resultant Architecture is known as segmented paging Architecture.

Virtual memory:

- Virtual memory gives an illusion to the programmer that a huge amount of memory is available for executing programs greater than the size available at physical memory
- Virtual memory is implemented through Demand paging

Demand paging:

→ It implies loading the pages on demand basis

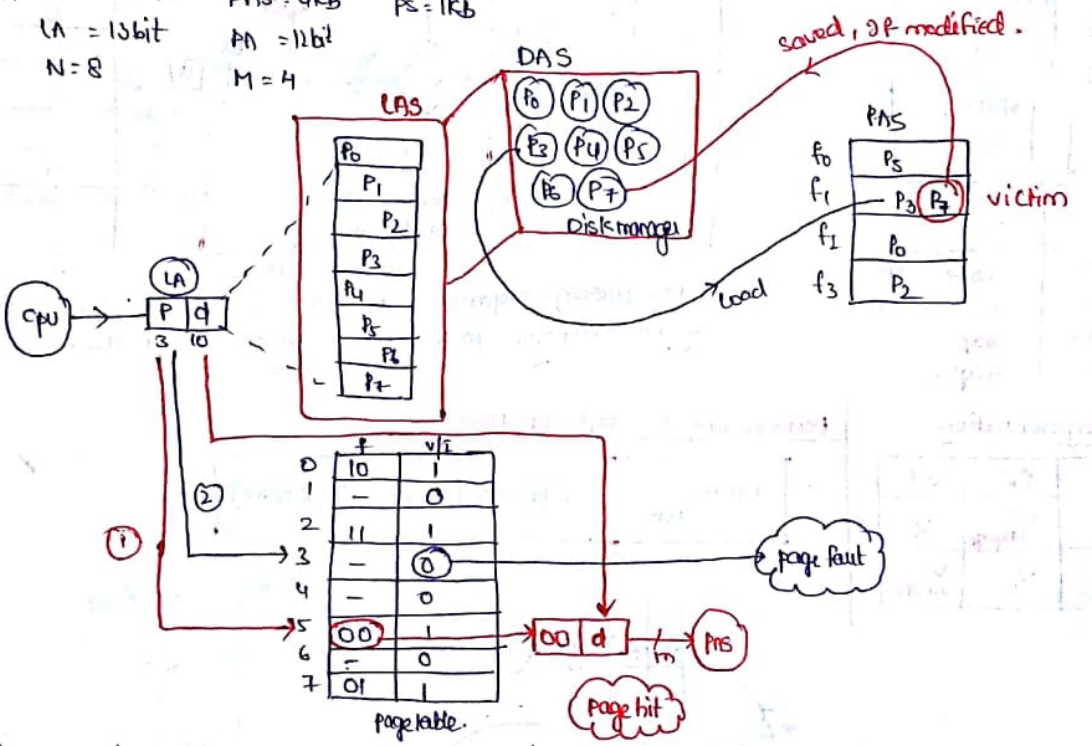
→ program is assumed to be stored on disk in the form of pages. These pages are initially loaded in the memory depending on the num of frames available.

→ The refered page in the VA may be present in the frame- (page hit) or may not be present (Page Fault)

→ If the page present in memory then it is resolved using page table as in the case of simple page.

→ If the page is not present in the mainmemory then the virtual memory manager makes arrangement to bring the faulted page from disk to memory with a possible page replacement

Ex:- LAS = 8KB PAS = 4KB PS = 1Kb
LA = 13bit PA = 12bit
N = 8 M = 4



*) Demand paging

Pure DP:	Prefetch DP
Execution starts with Empty frames	Execution starts with non-Empty frames

*) If victim is modified copy then save the page to disk and load the selected page to memory

*) After replacement proces moved to ready Q and got scheduled and executed further

If page fault occurs
↓
Proces goes to block state
↓ Informs read page
Virtual mem manager
↓ Contacts
Disk manager
↓ should
Load requested page to main mem through DMA

If free space avail → Load

No free space → select victim by page rep Strategy

Performance of Virtual memory :

1) Effective memory access time :

MMAT → 'm'
PF service rate → 's'
PF rate → 'p'
Page hit rate → '1-p'

$$EMAT_{DP} = p*s + (1-p)\,m$$

Points to remember :

Technique	EMAT
Paging	2m
n-level paging	(n+1) m
Paging with TLB	$\alpha[c+m] + (1-\alpha)[c+2m]$
2 level Paging with TLB	$\alpha[c+m] + (1-\alpha)[c+3m]$
n level paging with TLB	$\alpha[c+m] + (1-\alpha)[c+(n+1)m]$
Segmentation	$\alpha[c+m] + (1-\alpha)[c+2m]$
Demand paging	$p*s + (1-p)\,m$

*) Segmented paging without TLB = 3m
*) Segmented paging with TLB = $\alpha[c+m] + [1-\alpha](c+3m)$.

2) Page replacement :

Page ref string = { set of successively unique pages refered in the given list of VA's }

Pr$_i$ <722; 715; 012; 334; 755; 854; 860; 011; 339; 964; 676; 55>
PS = 100 (p = 7, d = 22)

$p = \frac{VA}{PS}$	$d = VA \% PS$

Ref string : <7; 0; 3; 7; 8; 0; 3; 9; 6; 5>

length of ref string (L) = 10
Num of unique pages = 7

Every process demands 'n' num of frames

Frame Allocation policies :

n → num of processes
S_i → Demand of process P_i
D → Total demand (ΣS_i)

M : Frames available
a_i : Allocated frames to proces P_i

$M \le D$
$M \le \Sigma S_i$

Ex :- n = 5; <P_1 .. P_5> M = 40

	S_i	$a_i = \frac{M}{N}$ Equal Alloc	$a_i = \frac{S_i}{D} * M$ Proportionate Alloc	$a_i = S_i/2$ 50% rule
P_1	5 (wastage)	8	2	2
P_2	30 (starvation)	8	15	15
P_3	10	8	5	5
P_4	15	8	7	7
P_5	20	8	10	10
	D = 80	M = 40	M = 39	M = 39

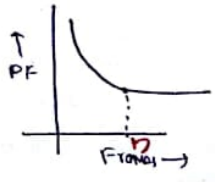
Page replacement techniques :

① FIFO :

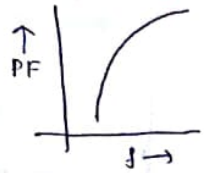
P	V/I	ToL
x	1	5

Criteria : Time of loading (ToL) → present in page table

FIFO curve



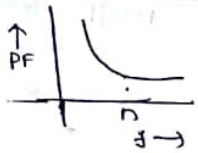
Belady Anomaly : ⟨1 2 3 4 1 2 5 1 2 3 4 5⟩



② Optimal page replacement :

In the event of page fault select that page as a victim which will not be used for the longest duration of time in future references.

Optimal PR curve :



- Practically non-Implementable
- Benchmark

③ Least recently used (LRU) :

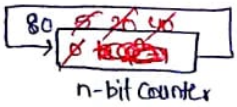
Criteria : Time of reference.

LRU Implementation :

→ In most cases LRU is very close to optimal page replacement

Implementation

Counter method

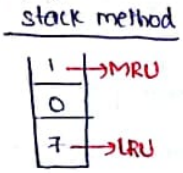


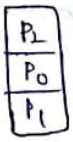
CPU cycles

	TOR
MRU ← P_i	80
LRU ← P_j	20
P_k	40
P_L	X → PF

∵ It may fail bcoz it can count upto 2^n clocks

Stack method





3 pop } for order.
2 push
1 push — for pages

If no page fault also the stack should be updated

3pop
3push

Worst case (bottom) 2n operations | Best case (Top next Element) 4 operations

∵ More stack overhead.

LRU approximations :

① Reference bit :

Prev Epoch					
e_1	e_2	e_3	e_4	e_5	e_6 (current Epoch)

Criteria : R ← 0 → Page not refered so far during present Epoch
1 → Page refered atleast once within current Epoch.

Page table

	P	V/I	TOL	R	M
0	a	1	4	1	0
1	b	1	2	0	1
2	–	0	–	–	–
3	c	1	0	0	0
4	d	1	3	1	1
5	e	1	1	0	0
6	f	1	5	0	1

→ P_1 is victim

→ second chance.

→ P_5 is victim by second chance Algo

:) If all R's are '1' then reference bit method will fail

(2) Additional reference bits :

Current Epoch ↑

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
P_i	0	1	1	1	0	1	1	1
P_j	1	0	0	1	1	0	1	1
P_k	1	0	1	1	1	1	1	1

P_j is victim.

:) During every new Epoch these 8 bit reference should be left shifted which is an o/H.

(3) Second chance Algo :

- It is a FIFO based Algorithm
- It can suffer from Belady Anamoly

Criteria : Time of loading + R / FIFO. (If R's are 1)

*) P_3 is given second chance and make R to 0

P_5 is selected as victim

(4) Enhanced second chance Algo :

- Not recently used Algorithm

Criteria : R + M (Modified bit)

R → Refered/not, M → Modified/not

Prev Epoch R'	Prev Epoch M'	Present Epoch R	Present Epoch M	Priority
		0	0	I
1	1	0	1	II
		1	0	III
		1	1	IV

:)
*) If we move from current Epoch to new Epoch we make all reference bits to '0'

(iv) Most recently used :

Criteria : Time of reference

(5) Counting Algorithms

- least frequently used (LFU)
- Most frequently used (MFU)

Tie breaking rule → FIFO

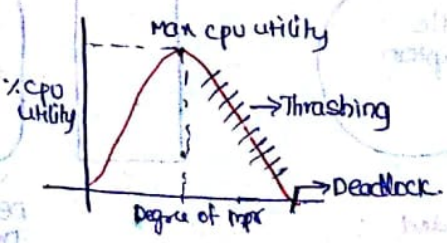
least frequently used: less count value

Most frequently used More count value.

Thrashing :

→ Excessive / high paging activity → Effects → Part of servicing a pagefault by VMM

- under utilization of CPU



Reasons for Thrashing :

1) lack of memory (frames)
2) Degree of Multiprogming

Control strategies of Thrashing :

- **Prevention :**
  → Controlling degree of mpr
- **Detection & Recovery**
  - **Detection :**
    - low cpu utilization
    - High degree of mpr
    - High paging disk utilization
  - **Recovery**
    - Decrease degree of mpr through suspension of process.

Other reasons for thrashing :

1) **Page replacement policy** : Bad replacement Algo
2) **Page size** : It Influences 1) IF
   2) Page table size overhead
   3) Thrashing → Small pages more PF rate
3) **Programming techniques** :

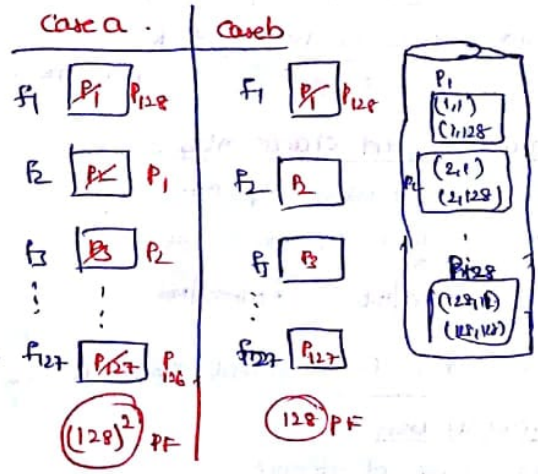
Int A [1--128, 1--128]

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

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

RMO, Ps = 128W
M = 127
Pure DP
FIFO.

	PF case (a)	PF case (b)
M = 127	$(128)^2$	128
M = 128	128	128
M = 1	$(128)^2$	128



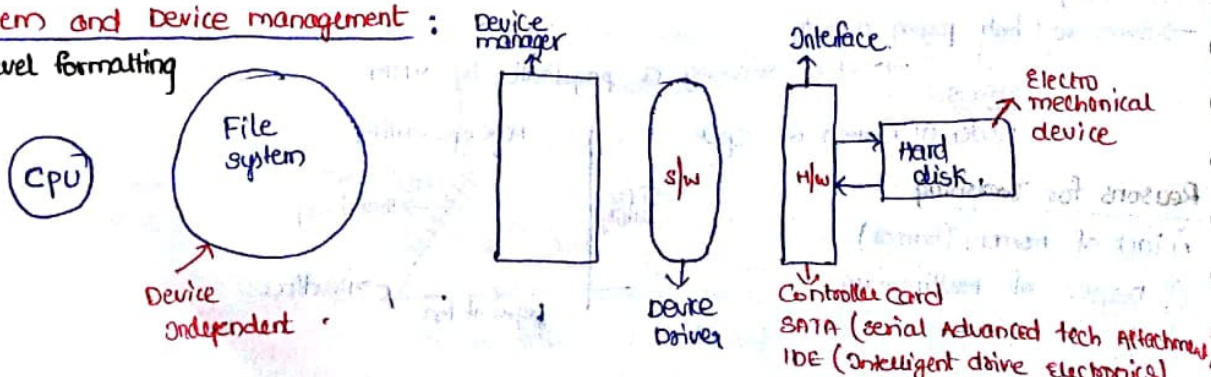
4) **Data structures** :
   - **Arrays**
     → Contiguous
     → less thrashing
   - **Linked list**
     - non-contiguous
     - More thrashing

*) **Searching**
   - **Linear**
     - less thrashing
   - **Binary**
     - more thrashing

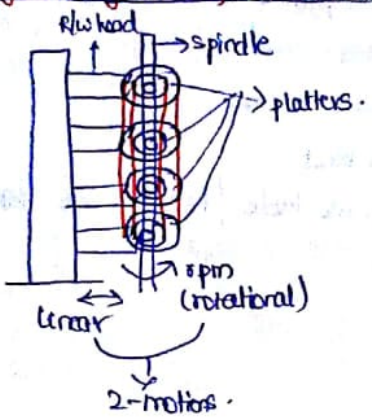
*) stack : less thrashing
   Hashing : more thrashing
   Pointers : more thrashing

File system and Device management :

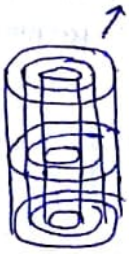
→ Low level formatting

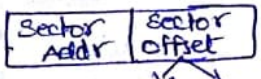


Physical geometry of Harddisk :



*) Num of cylinders = Num of tracks.





→ Good : ECC matches
→ bad : ecc doesn't match

Disk I/O time :

1) Seek time = $|x-y| * TTT$ (Track-Track time)

2) Rotational latency = $\frac{R}{2}$ ⇒ (universal)

3) Transfer time = $\frac{YR}{X}$ ⇒

Track size → 'X'B
Sector size → 'Y'B
Rot time → 'R'
1 Rot ⟶ X Bytes
? ⟵ Y Bytes
(TT) $\frac{YR}{X}$

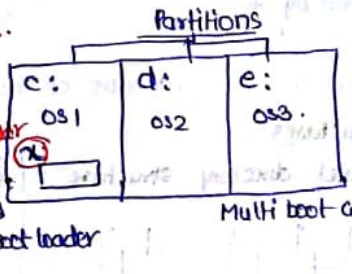
4) Data transfer rate = $\frac{X}{R}$ B/s

'X'B → R sec.
1B → $\frac{R}{X}$ sec.

Disk logical structure (High level formatting)

Volumes
- Primary → Bootable → (OS + Data)
- Extended / logical → Non Bootable → (Data)

Floppy drives. a: b:



MBR : master boot record

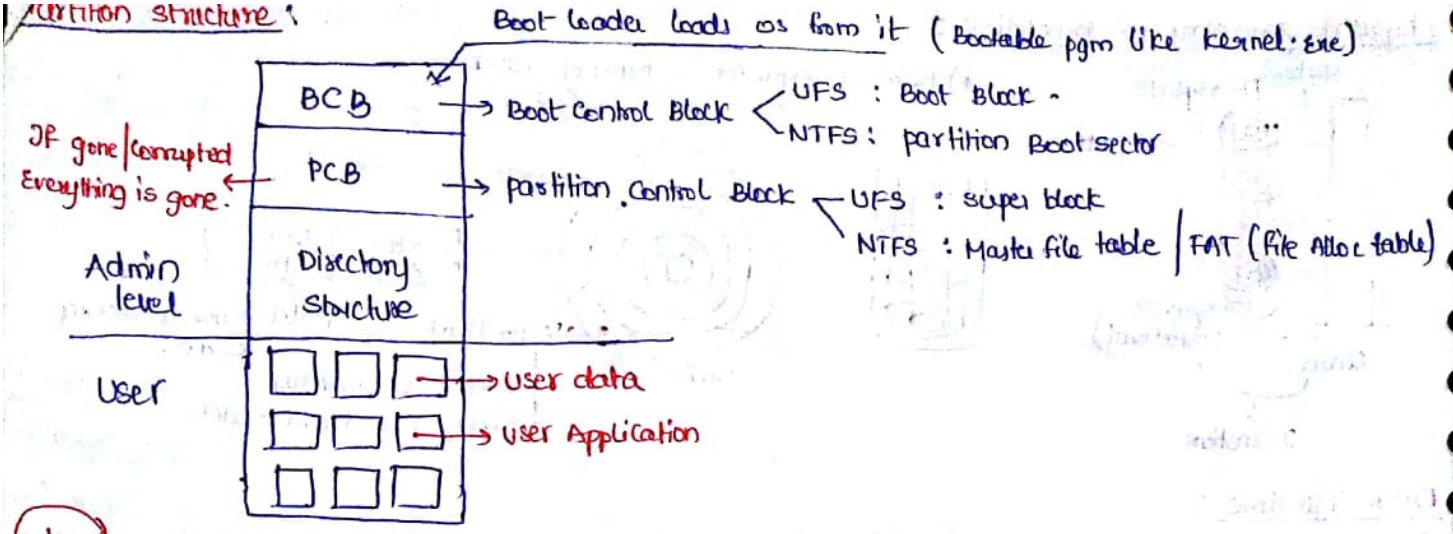
Booting process :

(H/w test) ← POST (Power on self test) — checks all H/w
↓
BIOS (Basic I/O system) — checks all I/p & o/p systems
↓
Bootstrap (Bootstrap is a pgm in ROM which loads MBR to main memory & gives control to Boot loader)
↓
Boot loader (provides option to select OS)

GRUB	LILO	NT LDR
Grand unified Boot loader (UNIX)	Linux loader (Linux)	New tech loader (Windows)

↓
shell pgm / GUI
↓
Booting completed.

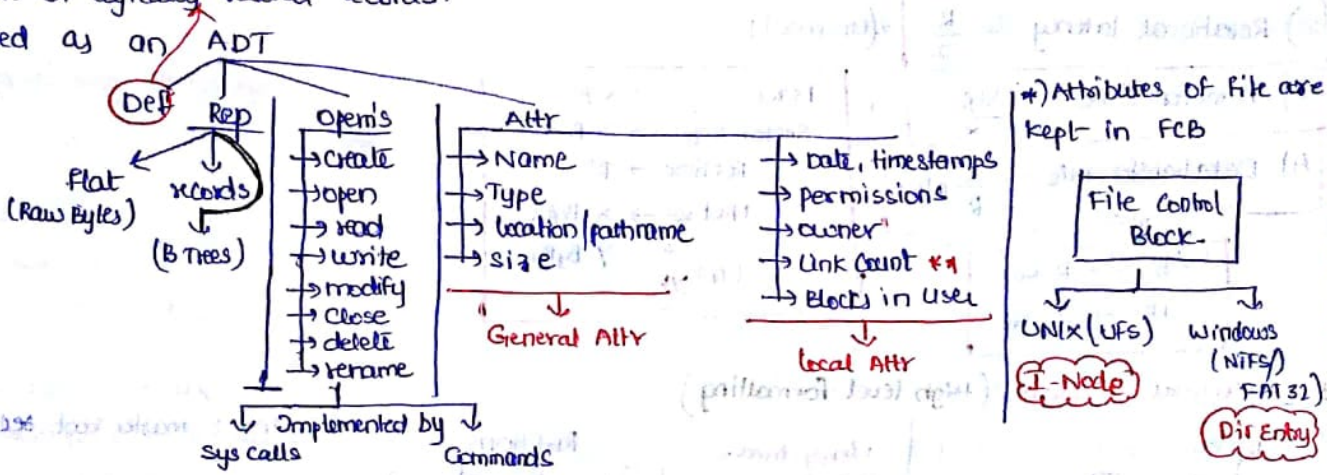
Partition structure :



Files & Directories :

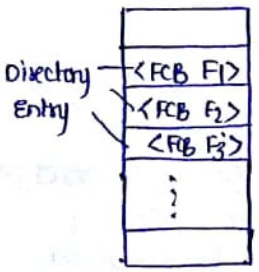
Collection of logically related records.

: viewed as an ADT



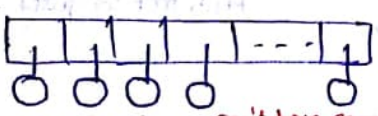
Directories : A special file which contains attributes of the file in a well organized structure.

Abstract view :



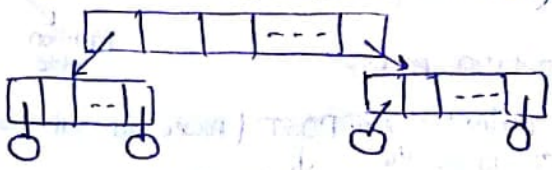
Directory structures :

① single level directory structure :

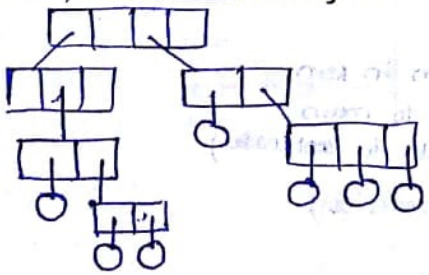


∴ More searching time, can't have same name for the files.

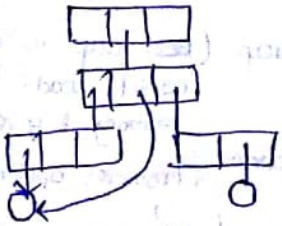
② Two level directory structure :



③ Tree structured directory :



④ Acyclic graph Directory structure :



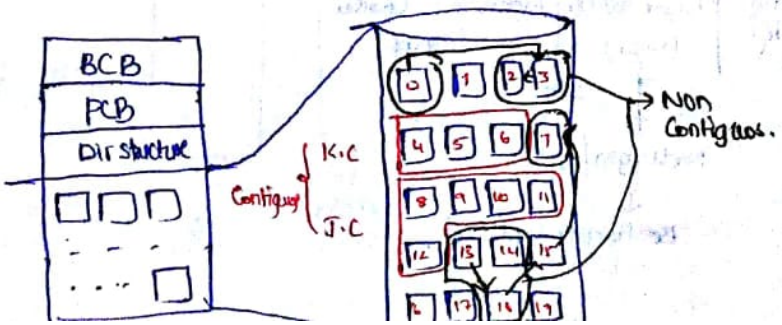
∴ File sharing ✓

Replication
- memwaste
- Inconsistent

Pointing
- No mem waste
- Imp by link counter

File Allocation methods :

- Contiguous Alloc
- Noncontiguous / Linked Alloc
- Indexed Alloc



DBA (Disk Block Addr) ⇒ 16 bits → 2^{16} ⇒ 64 K blocks
DBS (Disk Block size) ⇒ 1KB → Block size.
↓
$2^{16} * 1KB \Rightarrow 64MB$

Contigues Alloc :

Gen		Local	
FN	- - -	First DBA	Num of Blocks
K.C	- - -	4	3
J.C	- - -	8	5

Performance Issues ☺ Faster

① IF : ✓ (last block)
② EF : ✓
③ Increased File size : Difficult to Increase
relocation should be performed
Inflexible.
④ Type of Access : seq Access
Random Access.

Non-Contigues Alloc :

Gen		Local	
FN	- - -	First DBA	Last DBA
K.C		13	7
J.C		0	2

Performance Issues :

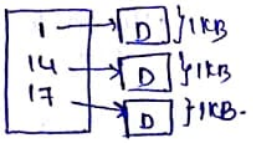
① IF : ✓ (last block)
② EF : ✗ (Never)
③ Increased File size : Flexible.
④ Type of Access : seq Access.

☹ - Slower
- Pointer Consumes disk space
- Vulnerability of ptrs leads to file truncation.

Indexed Allocation :

Gen		Local
FN	- - -	Index Block
K.C	- - -	16
J.C	- - -	19

16) Index Block



Unix Directory Implementation and Allocation method :

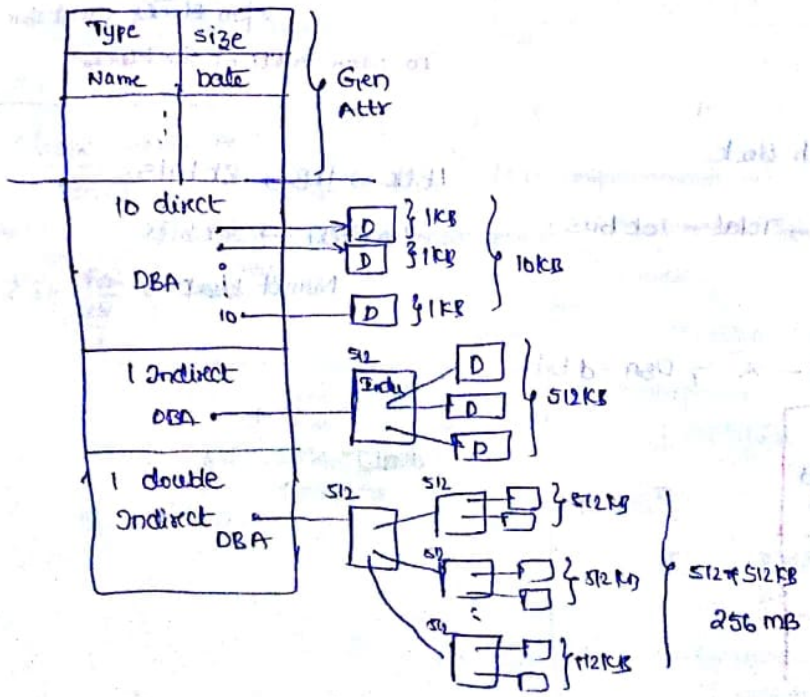
FN	Inode
K.C	23

DBA : 16 bits
DBS : 1KB

Max file capacity : $2^{16} * 1KB$
: 64MB.

Worst Inode
File size : 10 KB + 512 KB + 256 MB
: 256.522 MB

Inode :



Dos / windows Directory structure :

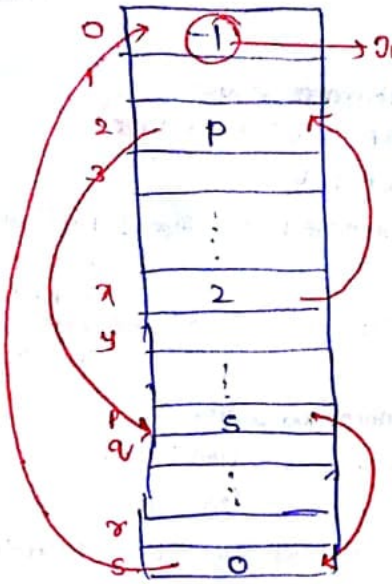
	Gen	Local
FN	...	First DBA
K.C	...	x

*) Num of Entries in FAT = Num of blocks

*) FAT entries holds Addresses DBA of next datablock in use by File

*) Fat Entry size = DBA bits.

File Alloc table (FAT) / Master File table



(Tabular, linked Allocation)

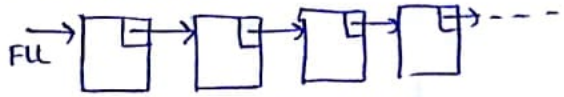
Disk free space mgmt :

DBs = 1KB, DBA = 16 bits, Disk size = 20MB, Blocks = 20k.

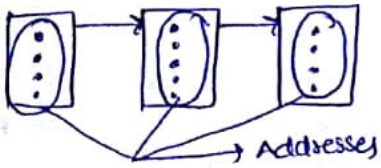
Free space Allocation techniques :

① Free linked list (FLL) :

*) linked list of free blocks.



② free list :



*) Best policy to Allocate free blocks because no searching is req'd

1 Blk → $\frac{1KB}{16}$ → 512 Addr.

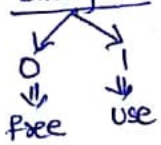
20k blocks → $\frac{20k}{512}$

→ 40 Blocks worst case

To store Addr of free blocks

③ Bit map / Bit vector :

- Associate a binary bit with each block



→ Total → 20k bits.

1 Blk → 1KB → 8k bits.

? Blocks → 20k bits

Num of Blocks → $\frac{20k}{8k}$ → 2.5

Points to remember :

Num of Blocks – B , Block size – X , DBA – d bits.

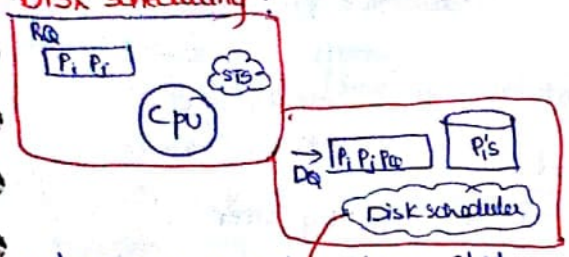
① Disk size – BX

② Max num of disk blocks – 2^d

③ Max disk size – $2^d * x$

④ $B \leq 2^d$

Disk scheduling :



*) processes present in block state.

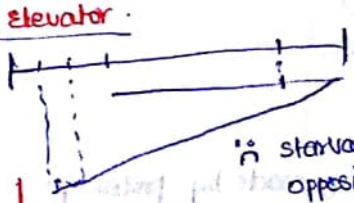
Objective.
To optimize num of seeks of a disk

Techniques :

① FCFS : First come first serve

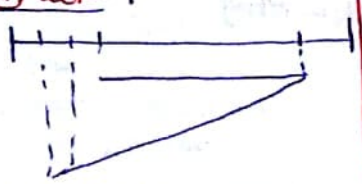
② Shortest seek time first (Closest Track next)
- optimal
- Time Consuming

③ scan / elevator :



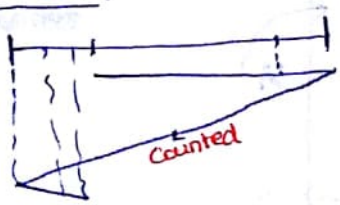
∴ starvation to process in opposite order

④ look :

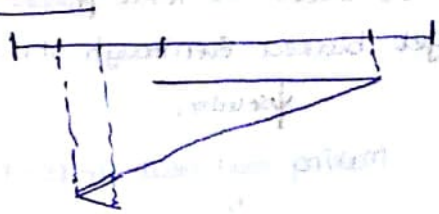


- Intelligent

⑤ C-scan :



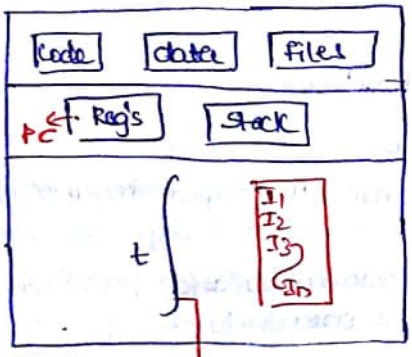
⑥ c-look



Leftovers :

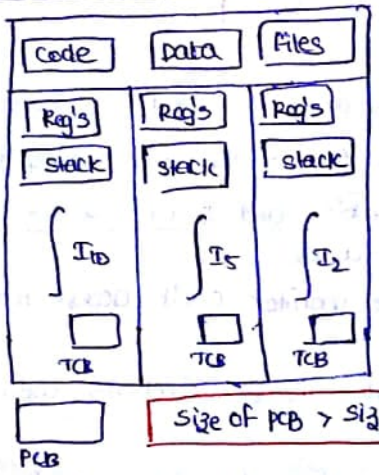
Threads & Multithreading :

Single thread process :



Indicates our pgm contains Branched Instructions.

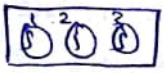
Multithreaded process :



Size of PCB > Size of TCB

Thread definition :
- Light weight process
- unit of cpu utility
- Active Entity.

☺) - Resource sharing
- Economical (Cost effective)
- |TCB| < |PCB|
  ↓
  Faster thread switching o/H than Context switching o/H.
  ↓
  more performance
- Computation speedup in Multicore Arch

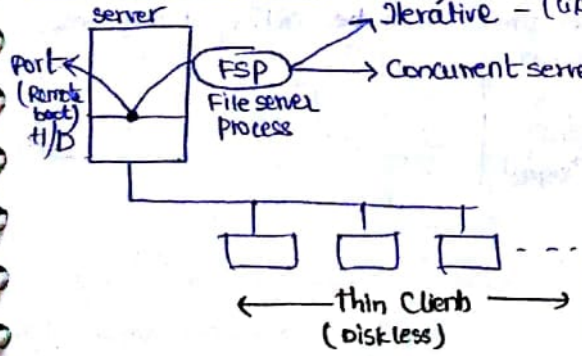


*) Booting

Local boot	Remote boot
→ pc's	→ on server → Ex:- Lab

Evolution of Threading :

Client server Arch



∴ starvation
Iterative - (like unipprogrammed os)

Concurrent server
- Multiprocess : Fork()
- + Threading

Concurrent server
- Multiprocess
  ↓ Imp by fork

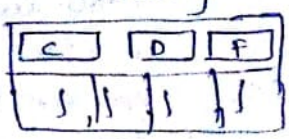
Schedule it using RR(TQ)
↓
Concurrency achieved
∴ Nostarvation

- Threading

∴ If k processes should run k copies of (Code data files) of created

↓ Sol

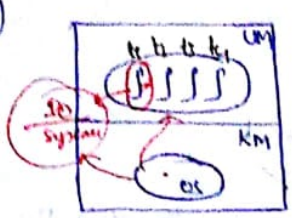
Threading



Types of Threads :

1) User level threads (Java threads, posix threads)
2) Kernel level threads → (portable as Interface for Unix)

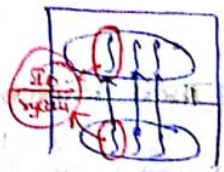
*)



In OS view the req made by process 'p' so, the OS blocks the total process therefore t4 t2 ts get blocked Even though it is active.

↓ solution

Making multithreads in kernel mode



User level thread :

- Thread mgmt can be done by user
- ☺ Flexibility
  Transparency
  Fastest Context switching (beeg of no mode shifting)
  (Horizontal switching rather than vertical switching.
- ☹

Monitors : (Hansen)

☹ Semaphore : If it is not programmed well it leads to deadlock.

→ Monitor is Implemented as an ADT in the programming language
↳ collection of procedures, variables and Data structures that are all grouped together in a special kind of module or package.

*) procedures that run outside the monitor can't access monitor internal variables (Abstraction) & Datastructures

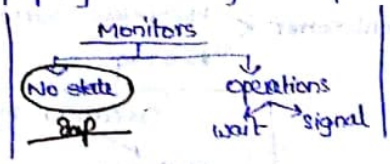
*) However they can active monitor member functions, which inturn can manipulate monitor Datastructures & internal variables.

*) Monitors also contain condition variables along with mutex lock

*) Condition variables are Associated with wait and signal opers which are used to meet the sync requirements of the application.

*) Monitors have an Important property that only one procen can be active at any time

*) Semaphores
↳ state (value) ↳ operations → up, down



Producer Consumer problem.

```
Monitor prod_Cons
begin
    Integer Count          → Internal var
    Condition Empty, Full  → Conditional var
    Procedure Enter        → procedure
    begin
```

```
    If (count == N) full.wait();
     Buffer[in] = itemp;
     in = (in+1) % N;
     Count = count + 1;
     If (Count == 1) Empty.signal();
End
Procedure remove  ⟶ procedure.
begin
    If (Count == 0) Empty.wait();
    itemc = Buffer[out];
    Out = (out+1) % N
    Count = Count - 1;
    If (Count == N-1) full.signal();
  End
  Init : Count = 0;
  End monitor.
```

```
Procedure producer
begin
   while(True)
   {
     a) produce item(&itemp);
     b) prod_Cons.enter;
   }
  end.
```

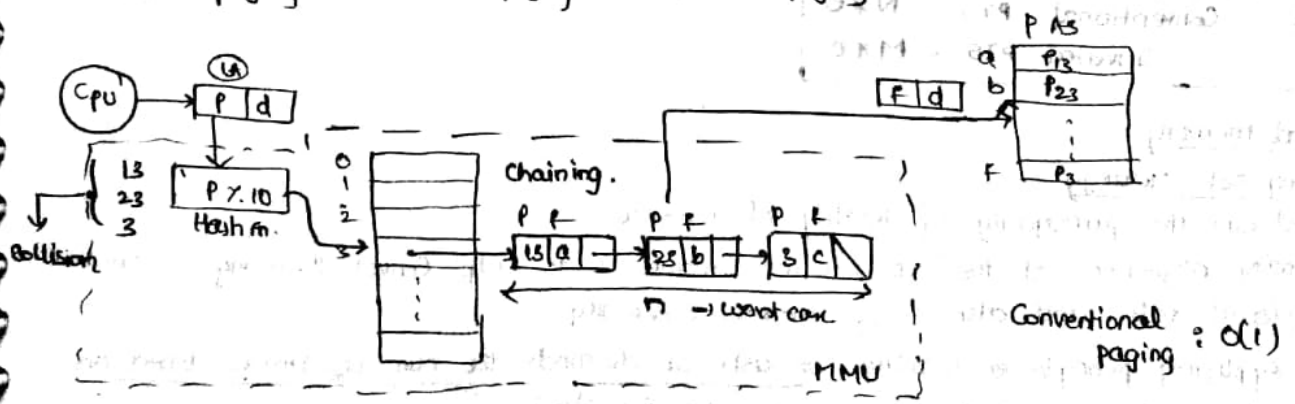
```
Procedure Consumer
begin
  while(True)
  {
    a) temp = prod_Cons.remove;
     b) process item(temp);
  }
```

Outside fn wants to access monitor so Implicitly mutex lock is applied so that one process can run on monitor.

Hashed paging:

To reduce page table size

- Multilevel paging
- Hashed paging
- Inverted paging



Conventional paging : O(1)

☺ space Efficient

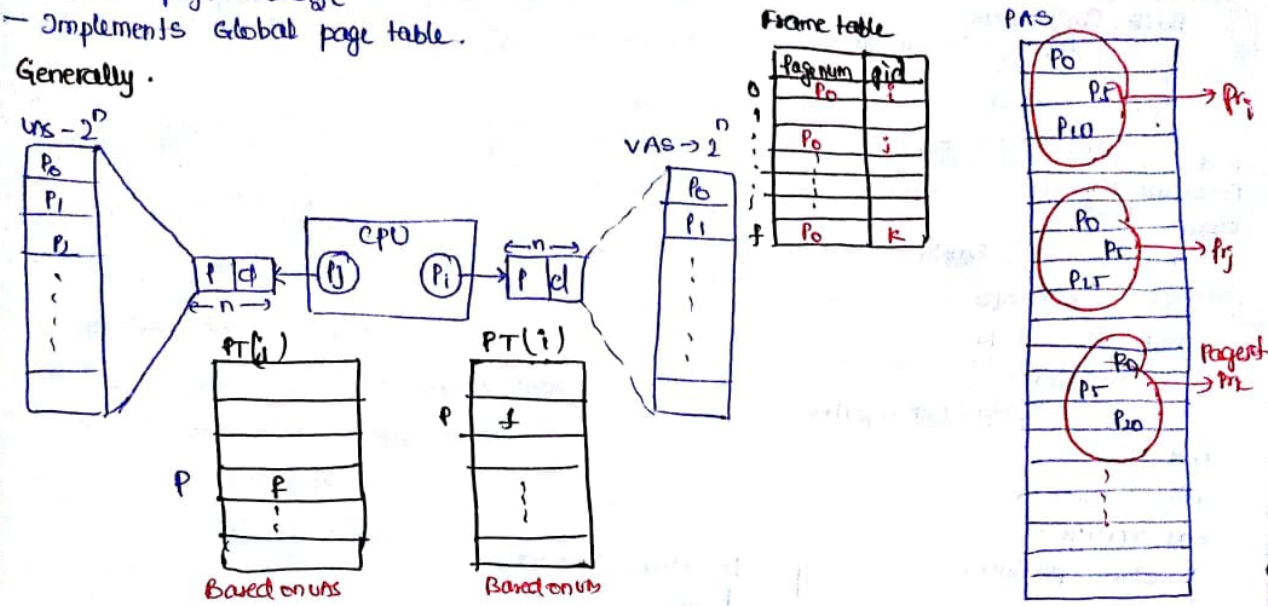
☹ O(n) → sequential Access.

Inverted paging:

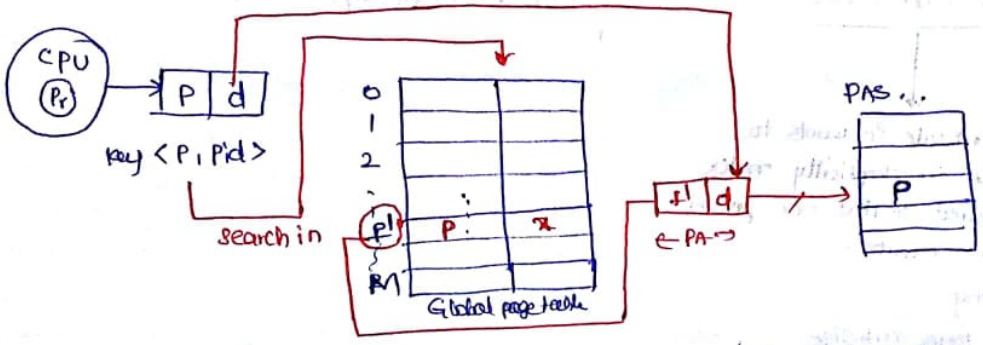
→ reduce page table size

– Implements Global page table.

Generally.



Address translation in Inverted paging:



∴ searching is O(n) | ∵ Space optimization is present in Inverted paging

Note: Conventional PTS = N * e

Inverted PTS = M * e

Virtual memory

Working set strategy:

→ Based on the principality of locality of reference

→ Primary objective of the working set model is to help control thrashing ie Controlling page fault rate and also utilize memory efficiently

→ By Applying principle of locality. It asks or demands the num of frames based on size of locality in which the program is executing

→ The size of locality is estimated by applying or by determining working set window size of the process at every time

Consider a pgm

```
10 KB  main()
       {
         f1();
       }
5KB    f1()
       {
         g1()
       }
```

```
20KB   g()
       {
         h()
       }
2KB    h()
       {
18 KB    scanf()
       }
```

Prog size : 55 KB
let PS : 1 KB
Num of pages = 55
↓ By frame Alloc (50% rule)
Asks 27 frames
↓
memory wastage may happen
↓
because of demanding 10 pages, 5 pages, 20 pages, 2 pages, 18 pages -- etc.

To Control this we dynamically allot frames.

At Run time :

Ref string Pr_k : 7,0,1,2,0,1,7,6,7,0,1,3,1,5,1,8,1,7,1,6,1,5

Δ = 10

33 35 38 39 42 40 33 34 35 36 38 ...

t1

Now,

Working set window - $WSW^{t_i}_k$ = { set of unique pages refered by process k from time t_i during past Δ references }

Here
Δ → Guess value
let Δ = 10

$WSW^{t_1}_{k\ \Delta=10}$ = {33, 34, 35, 36, 38, 39, 40, 42}

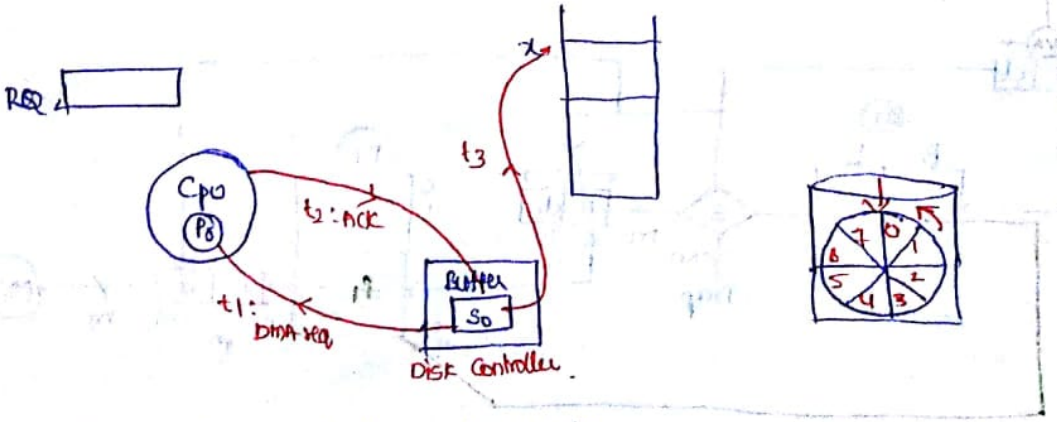
= (8) → It demands for 8 frames.

↓
Size of working set window
→ Always moving

*) let n = num of process
$WSWS_i$ = working set window size
$D = \sum_{i=1}^{k} WSWS_i$, M = Frames Avail

If D = M
→ No Thrashing
D < M
– No Thrashing
– more process can be included
D >> M
– Thrashing

Disk Interleaving :

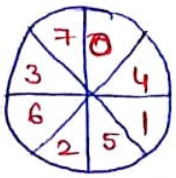


Total time for transfering = $t_1 + t_2 + t_3$
↓
Delay due to DMA o/H
↓
At this time the disk read some more sector and it is lost

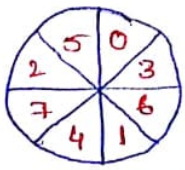
solution for the problem is.

h/w sol
Providing more buffers
ii) Cost tier

s/w solution
Interleaving
Single Interleaving
double Interleaving



2 rot read



2.75 rot read.

Segmented paging:

→ paging is used in segmentation to overcome the problem of external fragmentation in PAS
→ segment is divided into pages and pages are stored in frames of memory which later are accessed through page table.

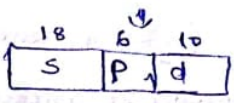
ie

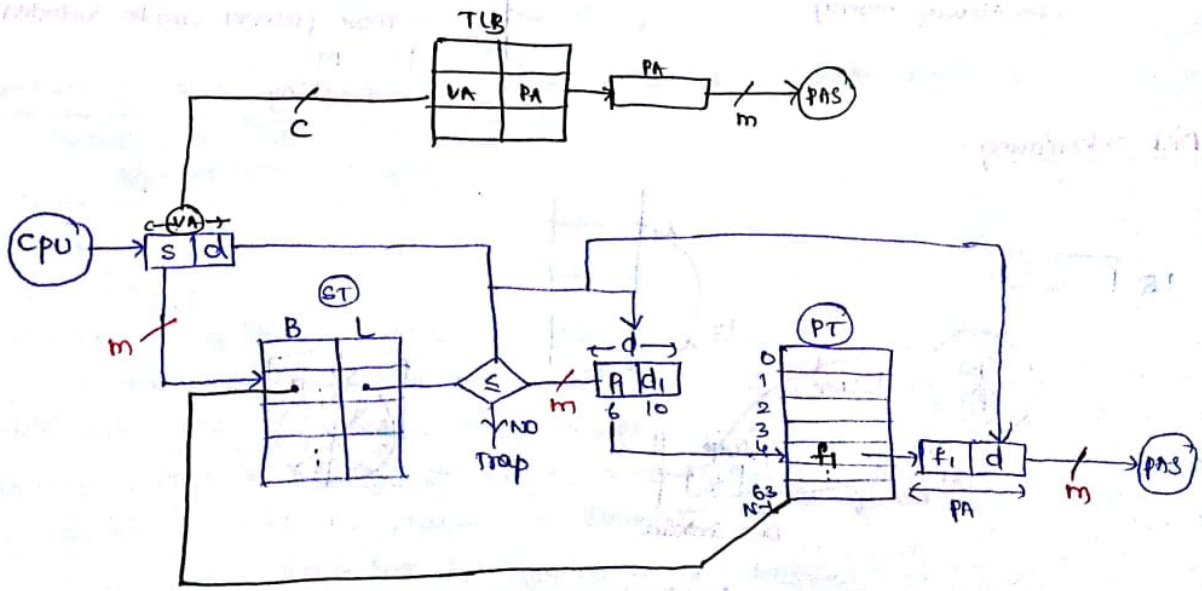
Every segment is refered through page table and page table is accessed through segment table.

Ex:- VA : 34 bits <18, 16> max Num of seg = 2^{18}, max seg size = 2^{16} = 64KB

let PS = 1KB So, for each seg, Num of pages = $\frac{64KB}{1KB} = 64$ → 6 bits

ie





*) EMAT without TLB = 3m

EMAT with TLB = $x[c+m] + (1-x)[c+3m]$

x — The End — x