OS Index :

Module 1 : (Introduction)

- Def of OS
- Goals - Domains - fns
- Types of OS : Disk tech
  user
  Poemption
- Arch requirements : I/O
  Memory
  CPU
- modeshifting, PSW
- fork system call

Module 2 : (Process Concepts)

- program vs process
- process def : ADT
  - rep
  - open's
  - Attr
- PCB, PSW
- states of process (DFA)
- Queues, Queuing diagram
- schedulers & Dispatchers
  Context switching
- CPU scheduling Algo : FCFS, SJF
  SRTF, LRTF, HRRN, Priority, RR
- Def like proces time
  TAT, WT, L, Th(μ)
  CPU scheduling fns & goals

Module 3 : (Proces mgmt)

- IPC → Inter
  → Intra
- Concurrency : Def
  types
  Condn's
  Mech
  Parbegin-parend
  fork & join
- Synchronization → pr Comp
  → pr Co-op
  - Need
  - Necessary Cond ⇒ CS, RC, P
  - Sync Mech Arch
  - Req for sync Mech ⇒ ME, Progress, BW
  - Mech
    - S/W ⇒ lock variable mech
      Strict Alternation
      peterson sol
    - H/W ⇒ TSL, SWAP
    - OS-based ⇒ Sleep & wakeup
      Semaphores
      Monitors.

Module-4 (classic IPC problems)

- Producer Consumer problem
- First reader writer problem
- Second reader writer problem
- Dining philosophers problem

Module 5 (Memory management)

- physical view of memory
- Compiler, preprocessor, Assembler, Linking
  Loading, Addr Binding, Binding times
  dynamic relocation
- Fns & Goals of mem mgmt
  (FMAD)
- Memory mgmt techniques
  - Contiguous
    - Buddy Sys
    - overlays
    - partitions
      - Fixed
      - Var
    - Partition Alloc policy
    - perf comparision
  - Non-Contiguous
    - Addres space
      - LAS org
      - PNS org
      - MMU org
    - ① Paging
      - Temporal issues
        - TLB fit
      - Spatial issues
        - Increasing PS
        - Multilevel paging
          hashed paging
          Inverted paging
    - ② Segmentation
      - paging vs segmentation
      - Segmented paging
    - ③ Virtual memory
      Demand paging - EMAT
      Working set strategy
      Frame Alloc policy
      Page replacement
      Algorithms
      - FIFO - Belady
      - OPR
      - LRU
        - Approximations
        - Ref, Add ref
      - Second chance
        - Enhanced
      - MRU
        - LFU
      - Counting
        - MFU

- Thrashing - def, Effects
  Control strategies → prevention
  → det & recovery
  Reasons - General
  Page rep Algo
  PTS
  Programming tech
  datastructures
  - Arrays
    Linkedlist
  - Linear search
    binary search
  - stack
  - hashing
  - ptrs

Module 6 : (File system & Device mgmt)

- Low level formatting & FS
- Physical geometry of Hard disk
- Disk I/O times - High level formatting
- Booting process - Logical structure
- PCB, Files, direc, dir structures - Disk interleaving.
  ↳ADT
- File Alloc methods (C, N, I)
- Inode, FAT
- Free space mgmt

- Disk scheduling Algo

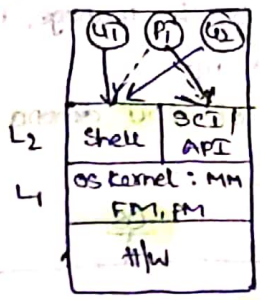
↓ ↓ ↓ ↓ ↓ ↓

FCFS SSTF scan look c-scan c-look

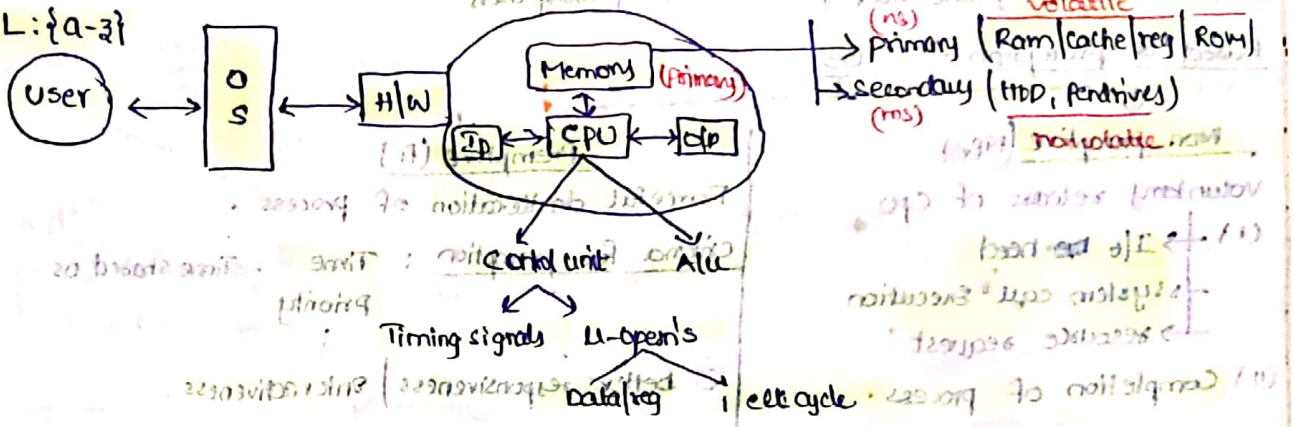
Operating systems :

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.



*) 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)

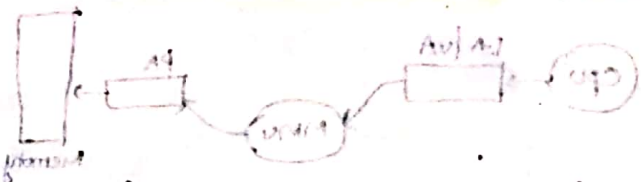
*) Primary Goals may change A/c to Domains.

→ 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

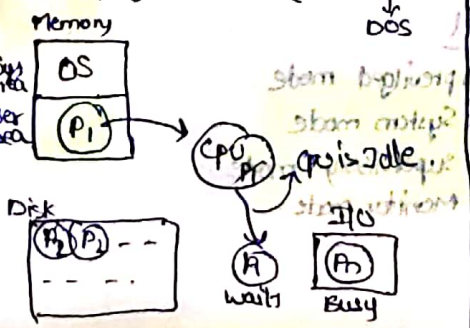
☺ Spooled device : A device for which tasks/Jobs are placed in Queue i.e., job Queue



Types of os :

Based on Disk technology

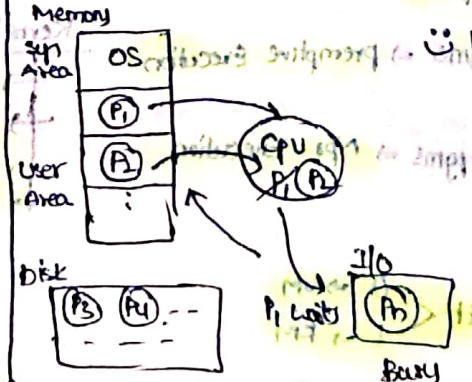
uni-programmed os (Ex: Batchos ↓ DOS)



∴ 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, more than 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

Multiuser 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 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.

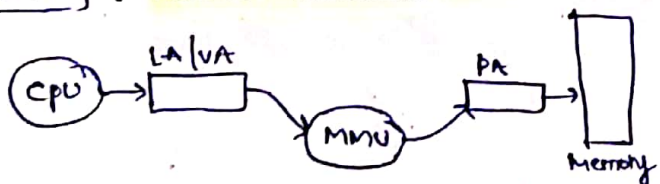
Mode of transfer (I/O)

Pgm Controlled IO
↓
Polling
✗ Mpr

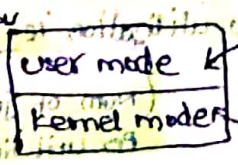
Interupt driven IO
↓
✓ Mpr
∴ processor load ↑

DMA
↓
✓ Mpr
∵ processor load ↓

② Memory : Address translation need because it provides security



③ CPU : Dual mode operation need

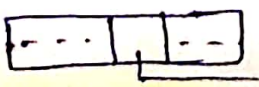


User pgms ⇒ premptive Execution

System pgms ⇒ Npr u Execution

Kernel
→ privileged mode
→ System mode
→ supervisory mode
→ Monitor mode.

PSW
↓
process status word



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 interrupt (ISR)

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

*) Interrupts
- H/w (raised by devices)
- S/w (Pgm driven)

PSW: Mode bit (1 = kernel, 0 = user)

Dispatch table

fork	x

System area → Executed without preemption → fork routine → last ins → Back to user mode

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");
}
```

Hi (1)
Hello (3)
Hello (2)
1. P
2. P
3. P
Both exec
Start from here

*) 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)	0	+ve (>0)
Error failure	child	parent

success

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

ret -1 → Error
ret 0 → child
ret +ve → parent

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)

Instructions + Data
- Static: → fixed size, → Load time
- Dynamic: → var size, → Run time

using of resources like cpu, IO, memory

Definitions of Process

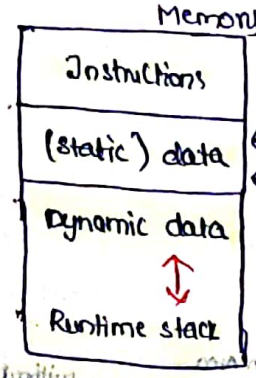
- → pgm under Execution
- → Instance of a pgm
- → Active Entity
- → schedulable unit
- → present in memory
- → unit of cpu utilization
- → locus of control of OS
- → Animated spirit of pgm

Developers view :

Process can be viewed as ADT

<Def ; Rep ; opers ; Attr>
Imp

Representation



Opers :

*) PCB is involved from creation to deletion

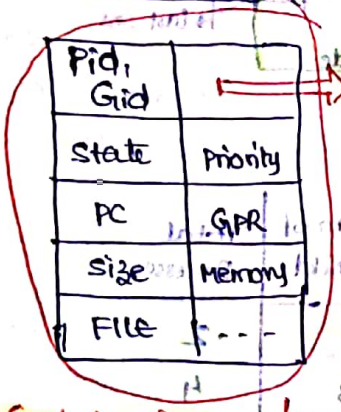
Create()
Schedule()
Dispatch()
Block()
Suspend()
Resume()
Delete()

Attributes

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

*) Size(B), Memorylimit, 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

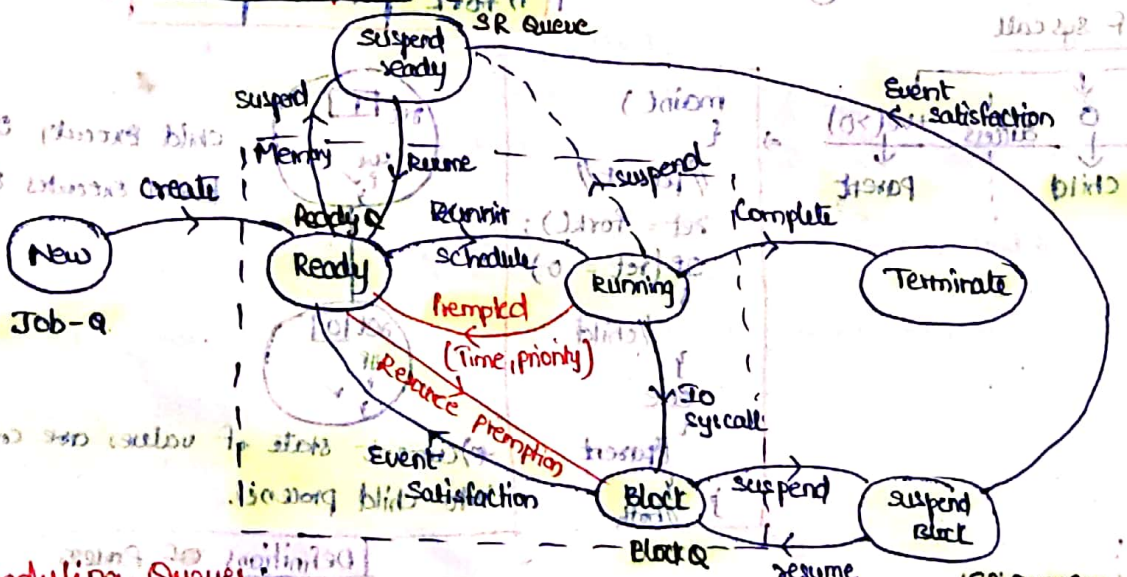
Running → ready ⇒ Nonprempted OS

→ 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 → P_i P_j P_k

RQ → PCBi → PCBj → PCBk

Block Q | Devices

DQ_1 → Pa Pb Pc | D_1

DQ_2 → Pd Pe | D_2

DQ_3 → Pf Pg | D_3

On-disk

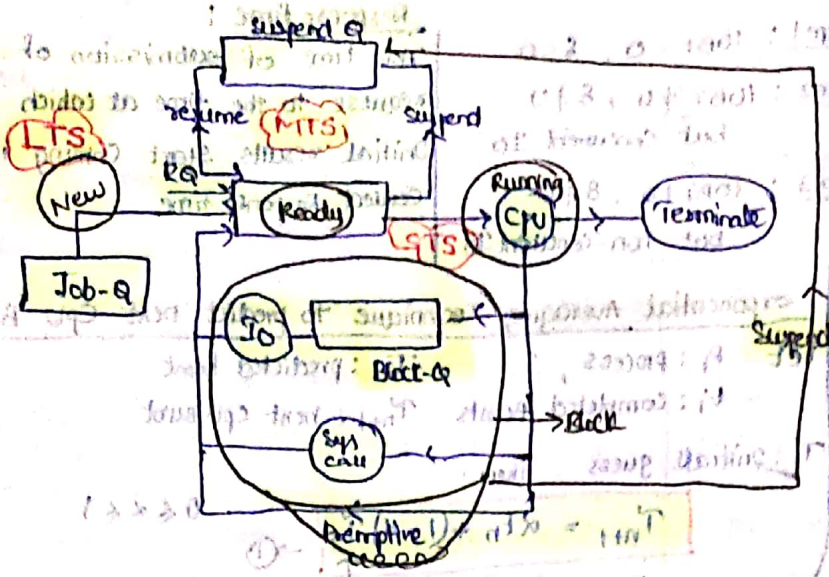
Suspend Q

→ list of suspended Pgm's

Input Q / Job Q

→ pgms ready to get loaded

State Queuing diagram :



Schedulers & Dispatchers :

Scheduler

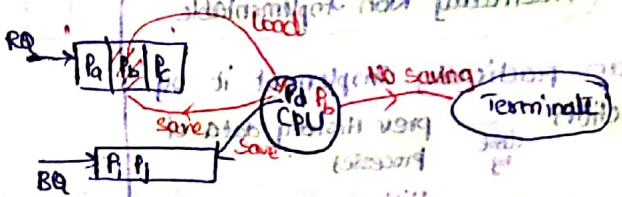
- **LTS** — Controls degree of multiprogramming
- **MTS** — suspensions & resumption → swapper
- **STS** — cpu schedular

Dispatcher :

→ carry out activity of Context switching

→ saving the PCB of preempted 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)

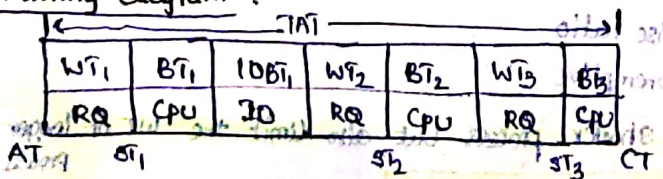
Min Turn Around time

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 :



$TAT = CT - AT$

$WT = TAT - [BT + IOBT]$

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

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

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

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

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

*) Schedulable Combinations

Non premptive — $n!$

Premptive — ∞

*) Throughput $(\mu) = \frac{n}{L}$ i.e. $n \to L$, $? \leftarrow 1$

CPU Scheduling Techniques :

① FCFS ;

Criteria : AT
Mode of opern : Non premptive.
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 :

Criteria : BT
Mode of opern : Non premptive
Conflict resol : AT + lower pid

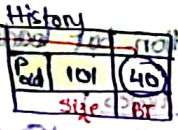
Optimal Algo

Performance of SJF :

- favours shorter process.
  - ∵ Max Throughput, Min TAT, WT (Avg)
  - ∴ Starvation to larger process

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



P: lookb
Predicted BT: 40

Exponential Averaging Technique to predict next CPU BT :

let P_i : process, τ_{n+1} : predicted burst
t_i : completed bursts t_{n+1} : next cpu burst
τ_1 : Initial guess Then.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \quad \rightarrow (1) \qquad 0 \le \alpha \le 1$$

$$\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$$

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

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

$$\tau_{n+1} = \alpha t_n + \alpha(1-\alpha)t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + \cdots \cdots \tau_1$$

Exponential Averaging.

③ SRTF : (Preemptive SJF)

Criteria : BT
Mode of opern : 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 : $\frac{1}{BT}$
Mode of opern : premptive
Conflict resol : lower pid $\frac{1}{n}$

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 : 0 ... 100 ... 1000
L ... H

⑦ Round Robin Scheduling : (Premptive FCFS) (Time shared OS)

Criteria : AT + TQ
Mode : Premptive

Performance of Round Robin:

Value of TQ

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

Points to remember:

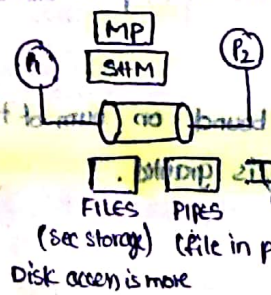
	Technique	Mechanism	Criteria	Mode of opern	Conflict resol
1	FCFS	First Come First serve	AT	Npr	lower pid
2	SJF (optimal)	Shortest Job First	BT	Npr	AT + lower pid
3	SRTF	shortest remaining time first	BT (Non Optimal)	Pr	lower pid
4	LRTF	longest remaining time first	BT	Pr	lower pid
5	HRRN	Highest response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

Process management:

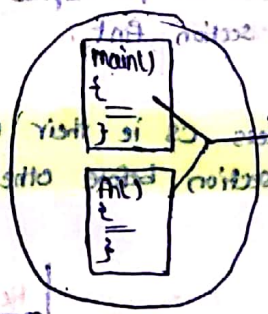
IPC; synchronization; Concurrency:

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

Inter process Communication:



Intra process Communication:



communicate through
1) Global variables
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 Deadlock**

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 :

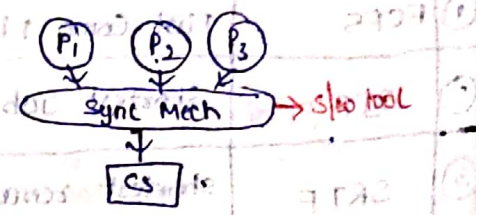
Neccesary Conditions :

① **Critical section** : (It is the part of the pgm where shared resources are accesed & 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 the finish their update.

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

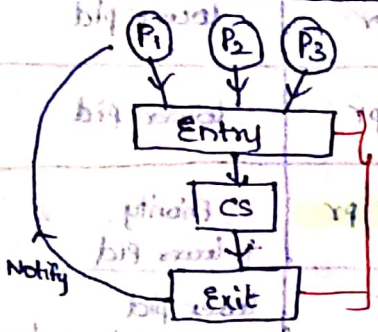
*) All 3 conditions are sufficient for sync problem

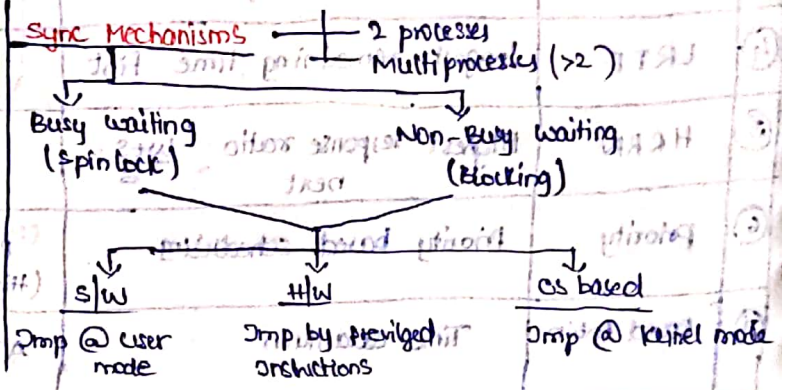


↓ sol

Architecture:

Synchronization Mechanism 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** :

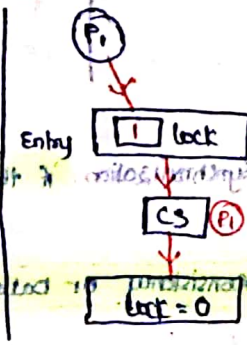
→ No process has to wait forever to access CS ie (there must be a bound on num of times a process is allowed to Enter Critical section before other process req is granted.

Lock variable sync Mech :

→ Busy waiting

→ s/w solution

→ Multiprocess Mech



Result

ME : ✗

Progress : ✓

Bounded waiting : ✗

Implementation : (HLL)

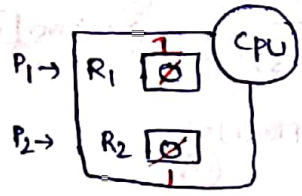
```
int lock = 0;
{
  while(1)
  {
    a) Non_CS();
    b) while ( lock != 0 );   Entry   (JNZ)
    c) lock = 1;
    d) <CS>
    e) lock = 0;              Exit
  }
}
```

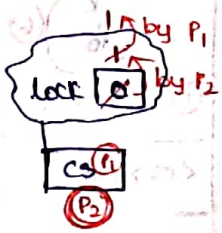
(LLL)

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

Mutual Exclusion : (X)

RQ | P1 | P2 | P3





(Loolevel lang)

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 : (✓)

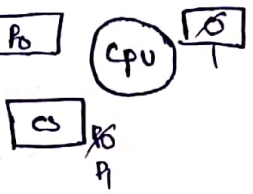
The process running in Non cs doesn't affect the interested process.

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 Meth :

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



Result :

ME : ✓

Progress : ✗

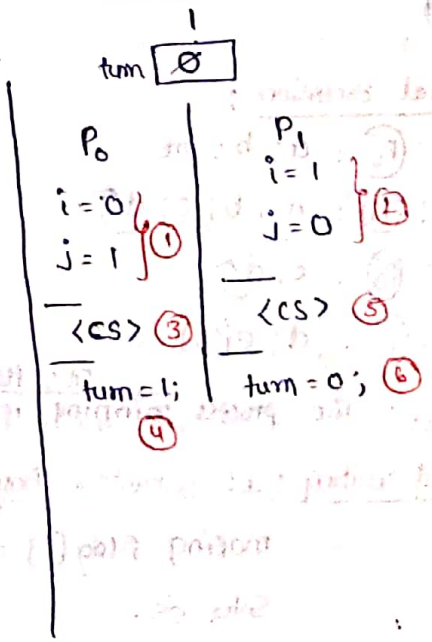
Bounded waiting : ✓

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
  }
}
```

loolevel

```
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
Exit
```



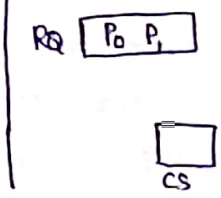
Mutual Exclusion : At any time only one process is present in CS.

Progress : If turn got to P_i and it is not interested 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

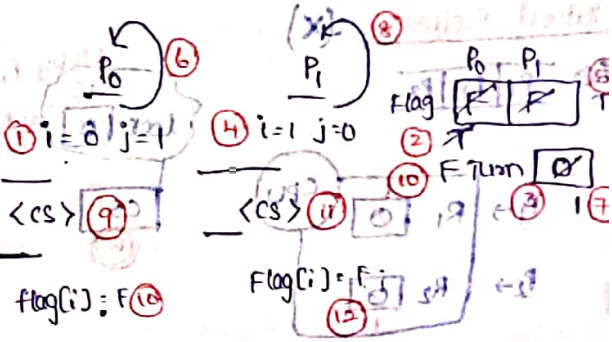


Result:
- ME: ✓
- Progress: ✓
- Bounded waiting: ✓

:(
- 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 interested process if it's flag value is F

Bounded waiting : It guranteed Bounded waiting because after accessing the CS we are making Flag[i] = T & 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;
  }
}
```

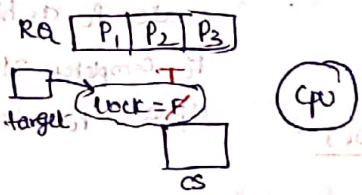
```
Bool TSL (Bool *target)
{
  Bool rv;
  a) rv = *target
  b) *target = TRUE
  c) return rv
}
```

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 accesing 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;
Entry  c) do {
       swap(&lock, &key);
     } while (key == TRUE);
    d) <CS>
Exit   e) lock = FALSE;
  }
}
```

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

Previlged Ins Executed w/o Premption.

Mutual Exclusion :

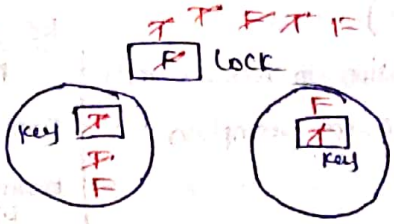
t_1 : (P1) : a, b ; pre

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

t_3 : (A) : c

t_4 : (P2) : c ; d ; e

t_5 : (P1) : c ; d ; e



ME ✓

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

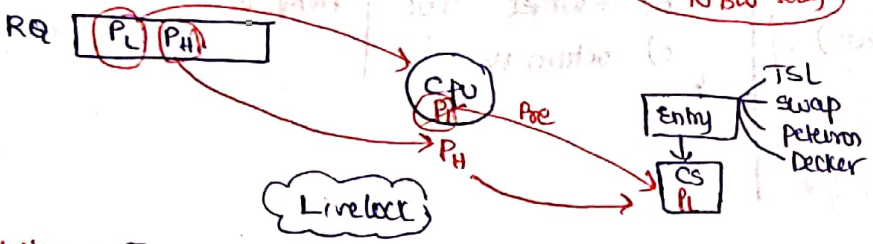
Bounded waiting : The process after accesing cs again enters into cs by swapping and making its Key values as False .

*| 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 N BW soln



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  L  C  J
         sleep();
    b) itemc = Buffer[out]
    c) out = out + 1 mod N
    d) Count = count - 1
    e) procen item(itemc);
    f) If (count == N-1)
         wakeup(producer);
  }
}
```

Procen Cooperation Deadlock.

Procen competition Inconsistency Datalocs

Scanned with CamScanner

*) Deadlock scenario in sleep & wakeup → Deadlock

RQ [P C]

Sleep Queue [P C]

CPU

N = 3

Count [0] 1 2 3

Buffer [x y z]

t_1 : C : L; C; J | ; pre

t_2 : P : (x, y, z) ; sleep()

t_3 : C ; sleep()

⑦ Semaphores :

→ Non Busy waiting
→ OS based sol
→ ~~Sequential~~ Multiprocess mech

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

*) Implemented as Abstract data type
- operations
  - Down : P : wait
  - up : V : signal / release
- state (value)

Atomic operations

Def of semaphores :

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

Based on range
- Counting semaphore : General : $<-\infty$ to $\infty>$
- Binary semaphore : Mutex : $<0,1>$

Counting sem
Bsem

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 :

S = 1 0 -1 -2 SQ (P2) (P3)

(P1) (P2) (P3)

Down(S)

CS P1; P2; P3

up(S) P1/P2, P3

*) If S.value = 4
- 4 → Down
- 3 → Down
- 2 → Down
- 1 → Down
- 0 → Down
- -1 → Blocked
- -2 → Blocked
- -3 → Blocked

Successfull Downs = 4

Blocked process = 3

SQ process = 3

Sleeping Queue
- FIFO → No starvation → Strong semaphores
- LIFO, priority → starvation → weak semaphores

*) S = 8 ; 10P ; 1V ; 18P ; 6V ; 3P ; 4V ; 5P ; 1V S = P − 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 ↓ unsuccess	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;   => list of all pcb's that are blocked while performing down unsuccessfully.
} BSEM;
```

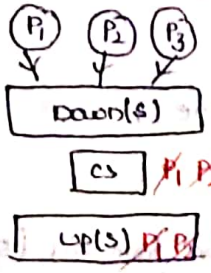
Implementation :

```
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 1

SQ P2 P3

P1 P2 P3

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 process is present in CS.

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

SQ (9P) (7P) (10P) (9P) (27P) (24P)

S=0 , SQ Contains 24 process.

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
bcoz 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 :

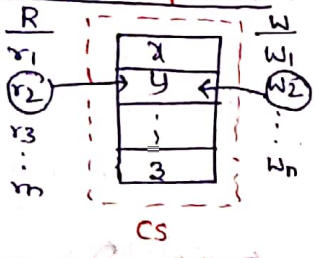
```
#define N 100
CSEM  Empty = N;   <Num of Empty slots in buffer>
CSEM  full  = 0;   <Num of full slots in buffer>
BSEM  mutex = 1;   <Used by (P) & (C) to access buffer as CS>
```

```
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 acces rc as CS>
BSEM db = 1 ;      < Used by (R's) and (W's) to access 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++;                       CS
    c) If (rc == 1)
         Down(DB)
    d) Up(Mutex) ;                 Exit
    e) <Read the DB>               CS
    f) Down(Mutex)                 Entry
    g) rc--;                       CS
    h) If (rc == 0)
         up(DB);
    i) Up(mutex) ;                 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)

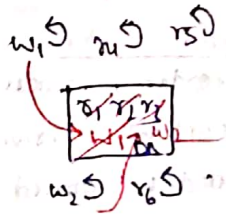
Case 2 : (R) and (W) arrived simultaneously (writer Allowed)

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

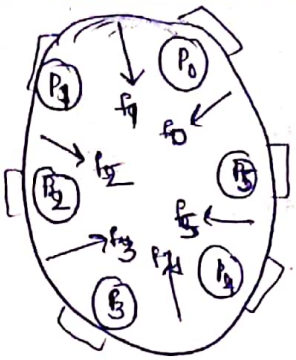


Solution :

```
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) up(read)
  g) <CS>
  h) Down(rmutex)
  i) rc--
  j) If(rc==0) up(DB);
  k) 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 :



$N \geq 2$

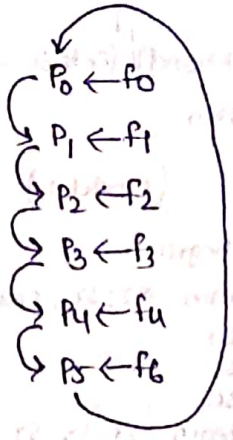
Max Concurrency = 3.

Non semaphore based soln :

1. (N-1) P ⇒ L-R
   1 P ⇒ R-L
2. odd p ⇒ L-R
   even p ⇒ 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);
    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 :

→ {

S_1 : a = b + c

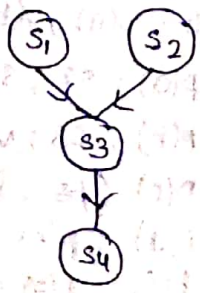
S_2 : d = e * f

S_3 : k = a + d

S_4 : l = k * m

}

Precedence graph



Concurrency types :

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

Concurrency Conditions :

$S \leftarrow R(S)$: Readset, $W(S)$: Writeset

$S_i \leftarrow R(S_i), W(S_i)$

$S_j \leftarrow R(S_j), W(S_j)$

Bernsteens Concurrency conditions

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 | maynot ϕ

Ex:- S: a+ = ++b * ++c ;

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

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

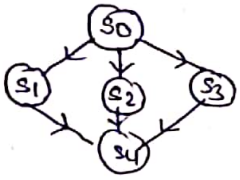
Concurrency Mechanisms : ① (Parbegin - parend) | (Cobegin - Coend)

② Fork and Join

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

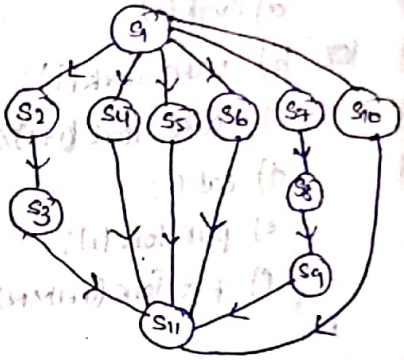
*) {
S_1
S_2
S_3
}

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



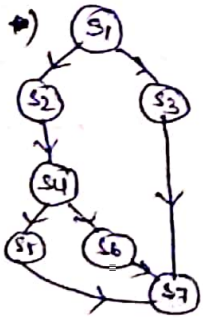
* S_1
Par begin
begin S2 ; S3 ; End
S4
S5
S6
begin S7 ; S8 ; S9 ; End
S10
Par End
S11 ;

(Model 1)



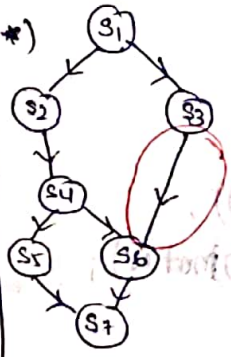
Model 2 :

*)



S_1
Parbegin
begin S2 ; S4
S3 Par begin
S5
S6
Par End
End
S3
Par End
S7

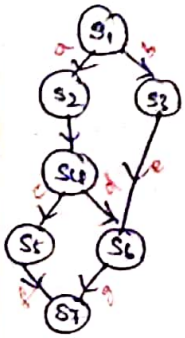
*)



⇒ Non-Implementable

But Implementable using Semaphores.

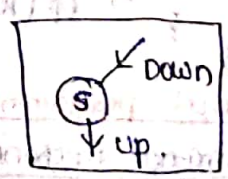
Par begin - parEnd with Semaphore :



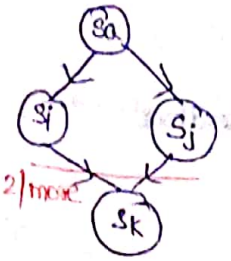
→ All process are blocked

Bsem a, b, c, d, e, f, g = 0 ; 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
Par End



Fork and Join :



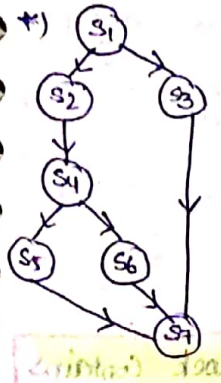
Syntax of fork :

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

Syntax of join : Join(Count).

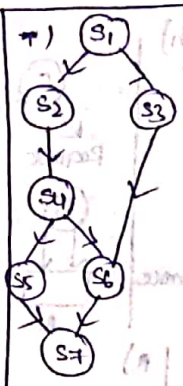
```
Join (int Count)
{
  Count = Count - 1;
  if (count ≠ 0)
  exit
  Else
   return;
}
```

*)




```
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
```

*)

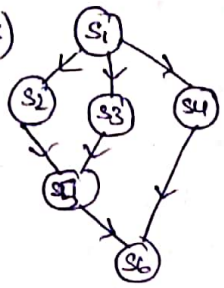



```
S1, a=2, b=2
Fork L1
S2
S4
Fork L2
S5
goto L3
L1: S3  goto L4
```

~~L2: S6 goto L3~~
~~L3: join(a)~~
~~L4: join(b)~~

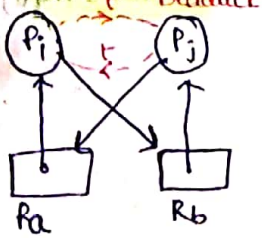
```
L4: join(b)
L2: S6  goto L3
L3: join(a)
S7
```

*)




```
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
```

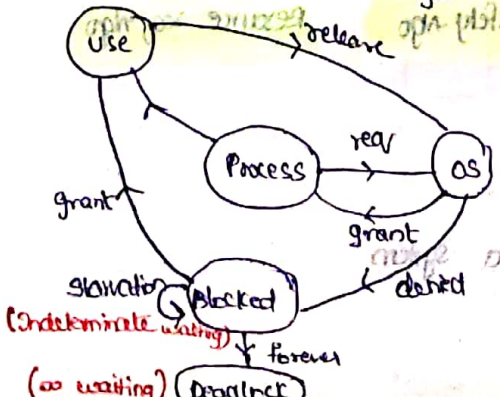
Deadlocks :



System model :

Consider n processes — m resources

- Single Instance
- multiple Instance

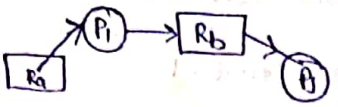


Characterization :

Necessary Conditions :

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

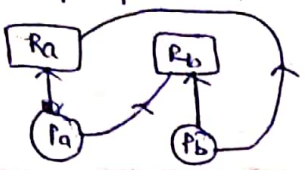
b) **Hold & wait** :



→ Every hold & wait doesn't imply deadlock.

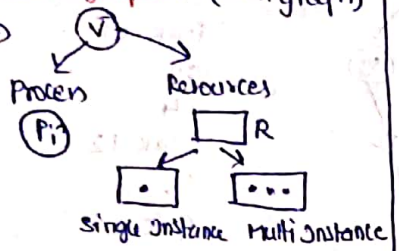
c) **No premption** : No premption of resources may cause deadlock

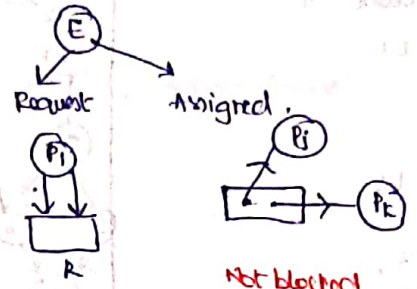
d) **Circular wait** :



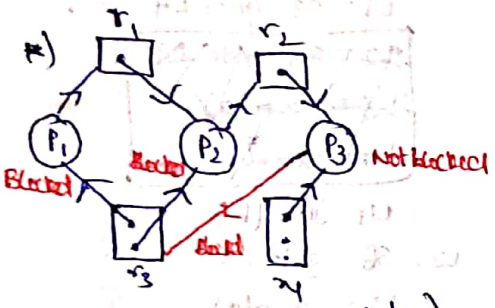
Resource Allocation graph : (Multigraph)

G(V, E) ⇒





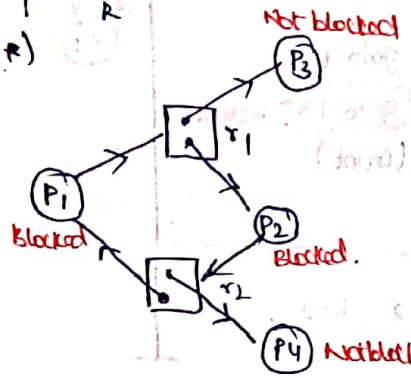
*)



$P_1 - r_1 - P_2 - r_2 - P_3 - r_3 - (P_1 | P_2)$ ⇒ 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 or

*) Every Deadlock 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.

Deadlock Avoidance : (Bankers Algo)

- Safety Algo
- Resource req Algo

→ Djikstra's

System State :

safe
unsafe

Datastructures that govern state of a system

1. n : num of process
2. m : num of resources

③ Max [1..n, 1..m]$_{n\times m}$ ⇒ Max[i,j] = k ie $P_i \xrightarrow{req\ max}$ 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[i,j] = d @ t ie $P_i \xrightarrow{req}$ 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	~~8~~
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	Run safety Algo
P_3	222	211	011	⟨532⟩
P_4	433	002	431	⟨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 :

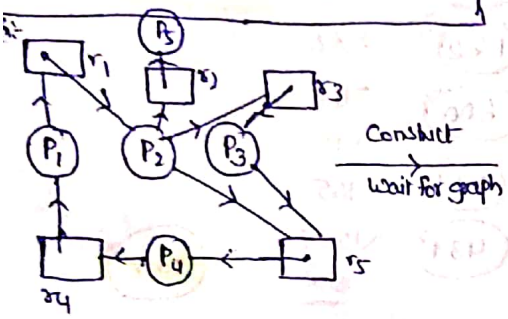
When to Apply ?

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

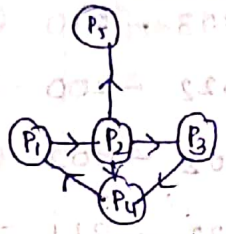
Detection when resources are single instance :

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

$P_1 - P_2 - P_3 - P_4 - P_1$ (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 A B C	Req A B C	Avail A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 3	0 0 0	3 1 3
P_3	2 1 1	1 0 0	5 1 3
P_4	0 0 2	0 0 2	7 2 5
			7 2 6

P_0, P_2 → not blocked
P_1, P_3, P_4 → Blocked
↓ majority
Run detection Algo

At t0 ; P_0 ; P_2 ; P_1 ; P_3 ; P_4 ⇒ safe

At t1 : P_2 req <0 0 1>

t1	Alloc A B C	req A B C	Avail A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

At t1 : P_0 ; ⇒ Deadlock

Blocked process : P_1 ; P_2 ; P_3 ; P_4

Deadlock recovery :

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

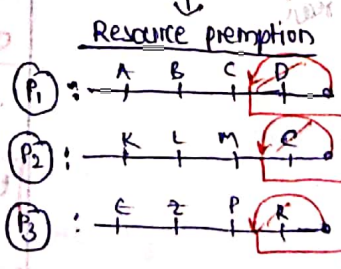
Deadlock Recovery :

- 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 preemption

*) All killed process need to restart from the beginning

*) Process Termination and Resource preemption suffer from starvation.

(Both suffer from starvation)



→ prempt latest resource of all process
→ Run detection Algo

Deadlock prevention :

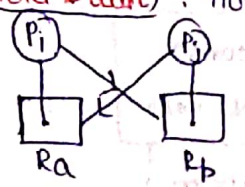
Dissatisfy one | more necessary conditions of the deadlock

→ Mutual exclusion
→ Hold & wait
→ No prempion
→ Circular wait.

1) Mutual Exclusion : (X)

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

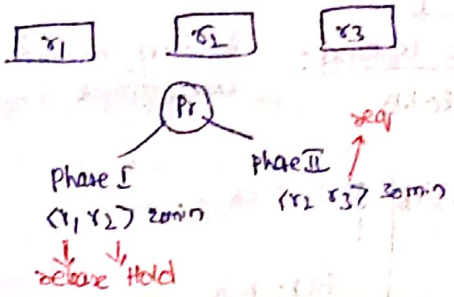
2) (Hold & wait) : Hold or wait



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

∴ under utilization of resources

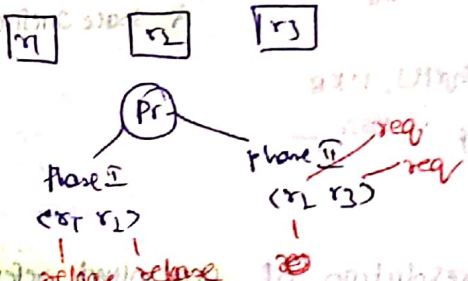
starvation.



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.

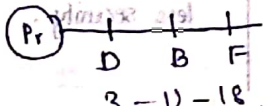
4) Circular wait :

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

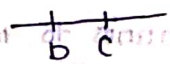
(ii) Never allows a process to req a lower num resource than the last one req & allocated.

Ex:-

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



prempt F & B.



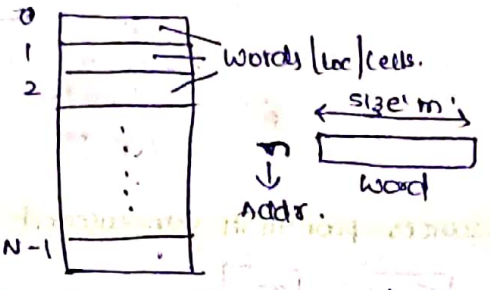
req for F & B → may | may not Available

∴ suffer from Starvation.

Prevention > avoidance > detection > ignorance

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 means reading the contents of the executable file ~~or pgm~~ containing pgm instructions into memory.

Loading:

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

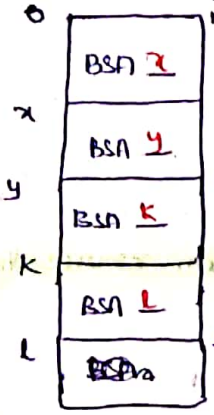
Loading
- Static
  - ☺ faster execution
  - ☹ space inefficient
- Dynamic (loading on demand basis)
  - ☺ space efficient
  - ☹ Time inefficient

Linking: It is resolution of unresolved references

- fn's
  - Library
  - User defined.
- var's
- objects.

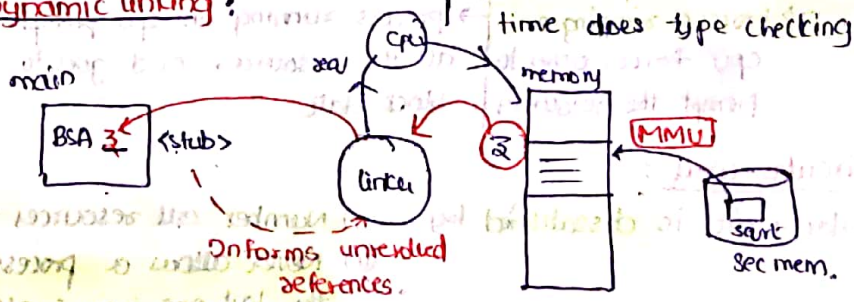
Resolution:

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

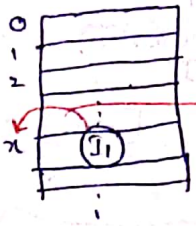
} difference.

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.



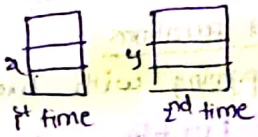
→ Decision by Addrn binding

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 Logical Addr
I_3 : 4
→ Loader at loadtime in memory with base addr 2000 loads
I_1 : 2000+0 : 2000
I_2 : 2000+2 : 2002 Physical addr
I_3 : 2000+4 : 2004
→ 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

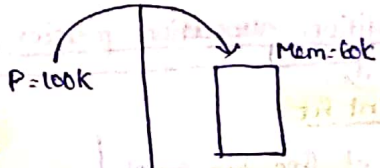
load time binding:
Compiler generates only logical offsets which are relocated to physical address by loader during load time

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 ↑

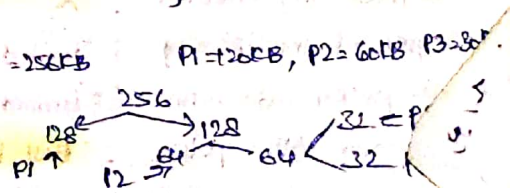
Memory management techniques:

Contiguos
↓
centralized Allocation
Ex:- partitions (2)
overlays (1)
Buddy systems (3)

Non Contiguos
→ decentralized / distributed Allocation
Ex:- paging
segmentation
segmented paging
virtual memory

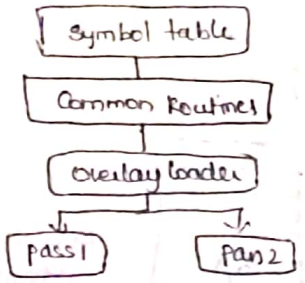
(1) **Overlays:**
Contiguos mem mgmt technique.

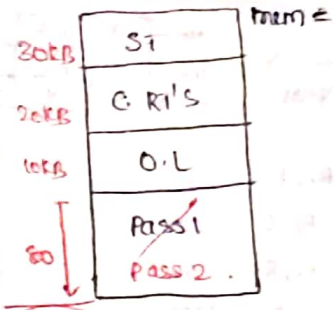
(3) **Buddy system:**
dividing the memory in closest powers of 2
Ex:- P=256KB P1=120KB, P2=60KB P3=30KB



2 pass Assembler :

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





20KB → 110KB → Min memory reqd.

Larger pgm to smaller mem.

Note :

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

2) partitions :

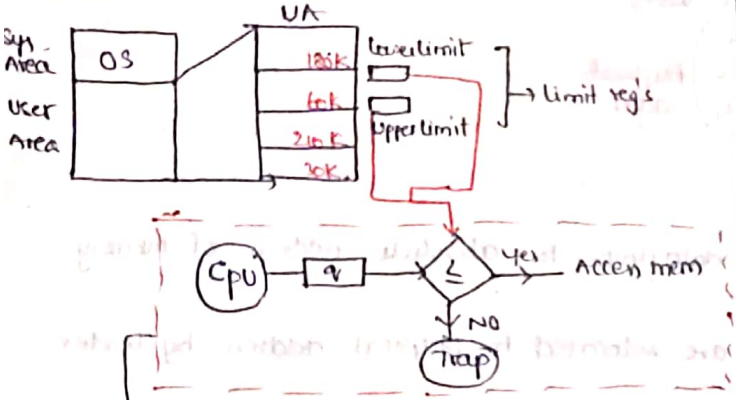
Fixed partitions

(multipgming with fixed tasks)

Rule : 1 partition = 1 pgm

→ Size is variable

→ Num of partitions are fixed.



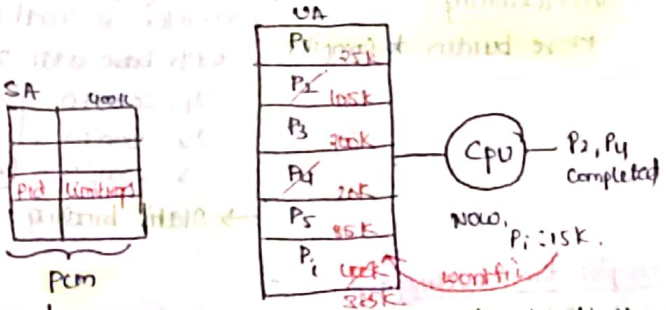
Provide security of non interfering of other pgms

→ 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 contiguos 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 paition 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 opems 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 partition
4) Max proces size runnable : limited to largest partition
5) superior 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 proces size)
3) Degree of multipgming : flexible
4) Max proces size runnable : flexible
5) superior Alloc policy : Worst fit

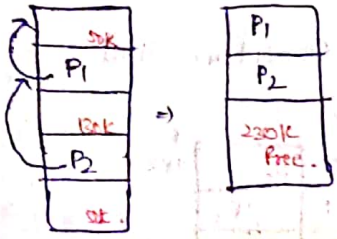
So, only problem in variable partition

↓

External Fragmentation

↓ solution

Compaction:



⇒ Compaction is supported by runtime binding

∴ Time consumption process ← bcoz of

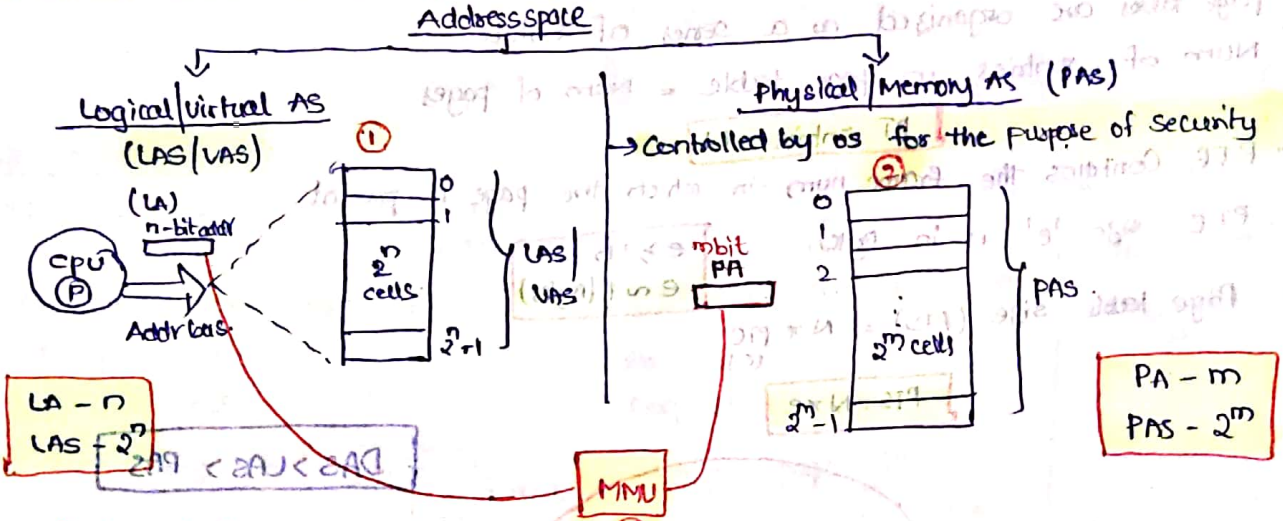
Non-Contiguous Allocation

→ 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

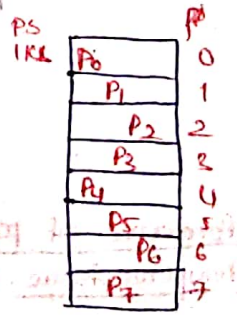


Study of Non-Contiguous allocation:

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.

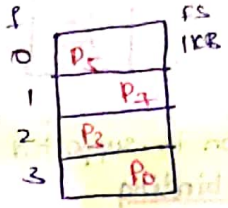
② organization of PAS :

Page frame / Blocks

- 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 :

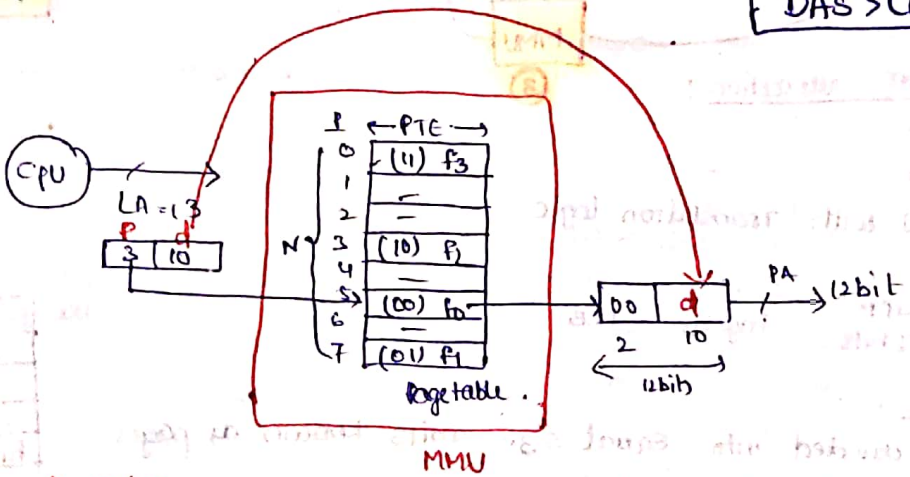
- Each process is associated with its own page table, which are stored in memory
- Page tables are organized as a series of entries.
- Num of entries in page table = Num of pages

  PT entries = N
- PTE Contains the frame num in which the page is present
- PTE size 'e' is in Bytes.

  e ≥ 1B | 2AU
  e ≈ f (Bytes)
- Page table size (PTS) = N * PTE (e)

  PTS = N * e

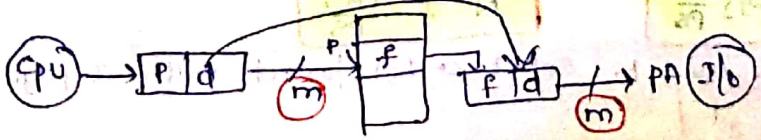
DAS > LAS > PAS



Performance of paging :

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

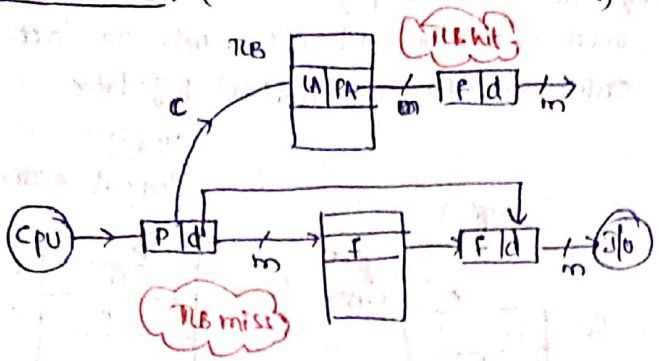
Let main memory access time be 'm'



EMAT = 2m

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

Paging with TLB: (Translation look aside buffer)



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

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

(ii) Special 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

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

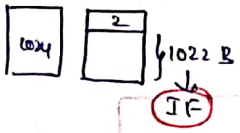
a) Increasing page size:

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

Draw
*) IF
Best Avg Worst
0 Page size/2 Page size

So, optimal page size should be selected.

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

⇒ $PTS = \frac{s}{p} * e$ $\quad IF = \frac{p}{2}$

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

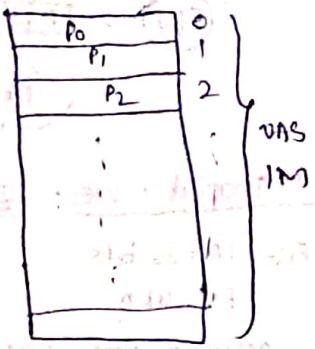
PS = 4KB

Let pages in PT = 1K

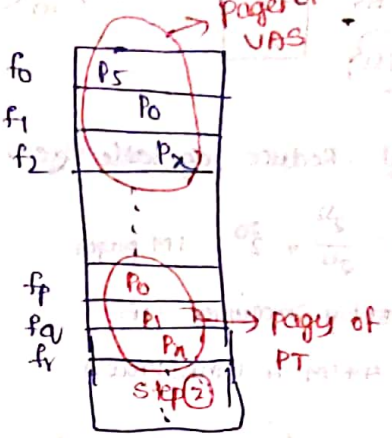
PS_{PT} = 1K

Step ①

Logical address space

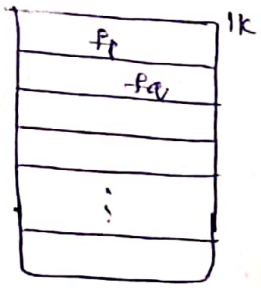


Step ②

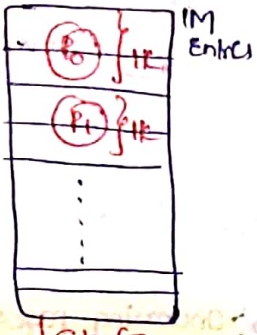


Outer PT

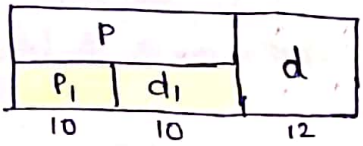
Step ③



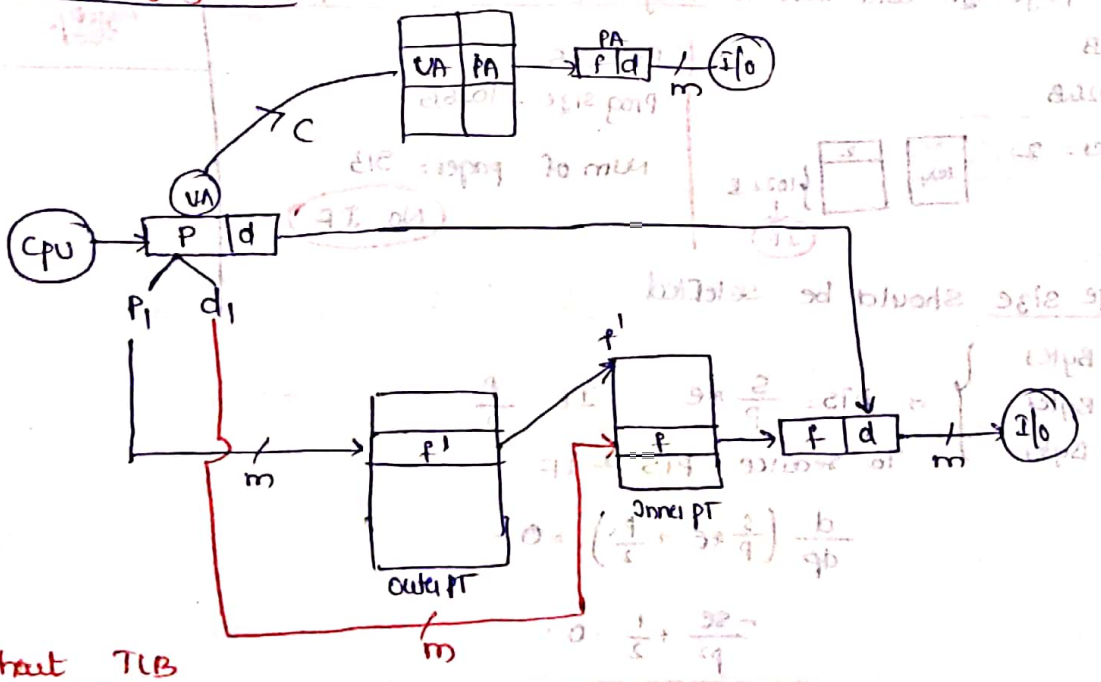
Inner PT Step ③



Address format :



Multilevel paging view :



Without TLB

$EMAT_{1lp} = 2m$

$EMAT_{2lp} = 3m$

⋮

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

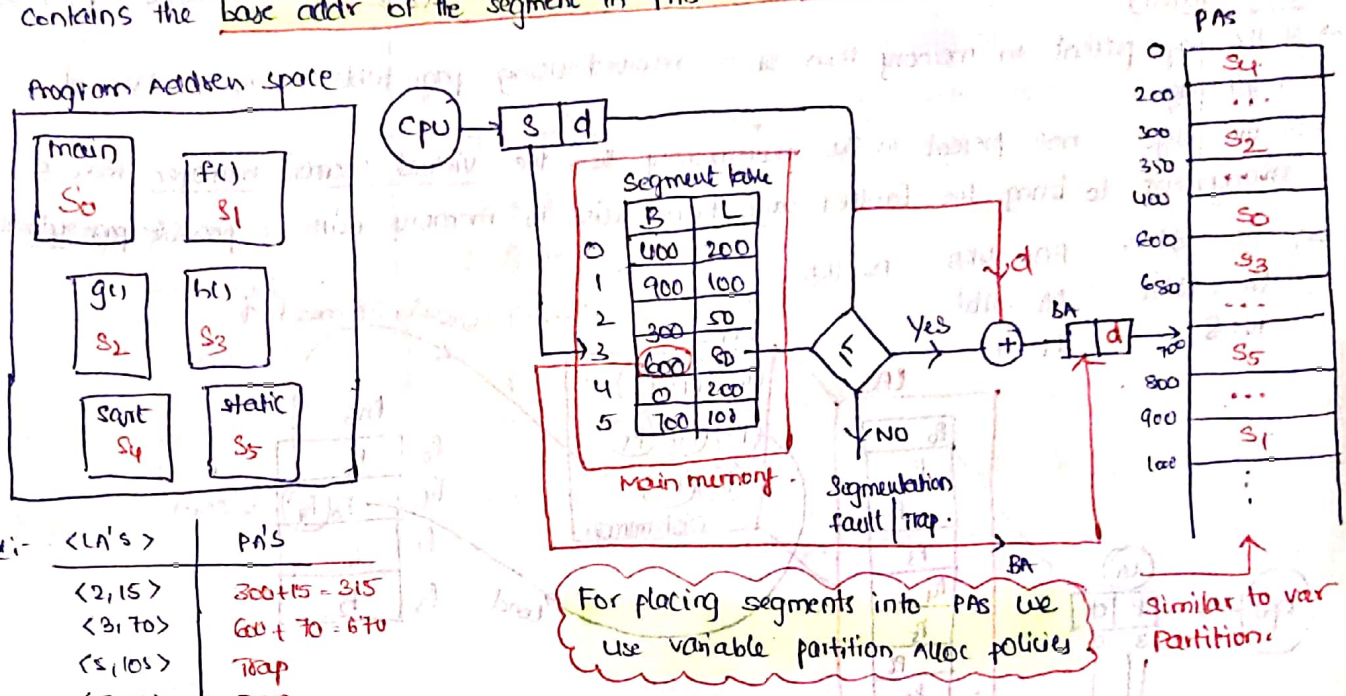
With TLB

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

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

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 addressing 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.



Ex:-

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

For placing segments into PAS we use variable partition alloc policies

Similar to var Partition.

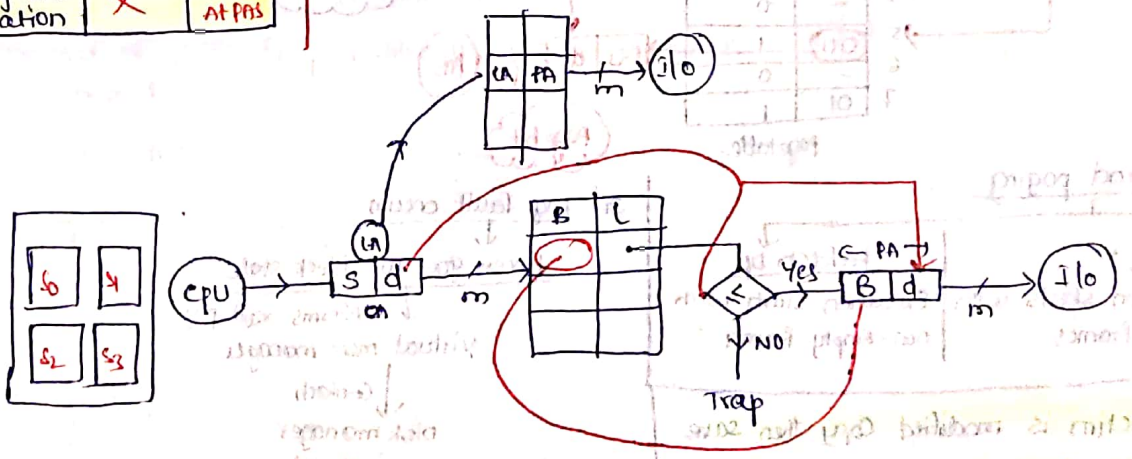
Paging vs segmentation :

	IF	EF
Paging	✓ last page	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.

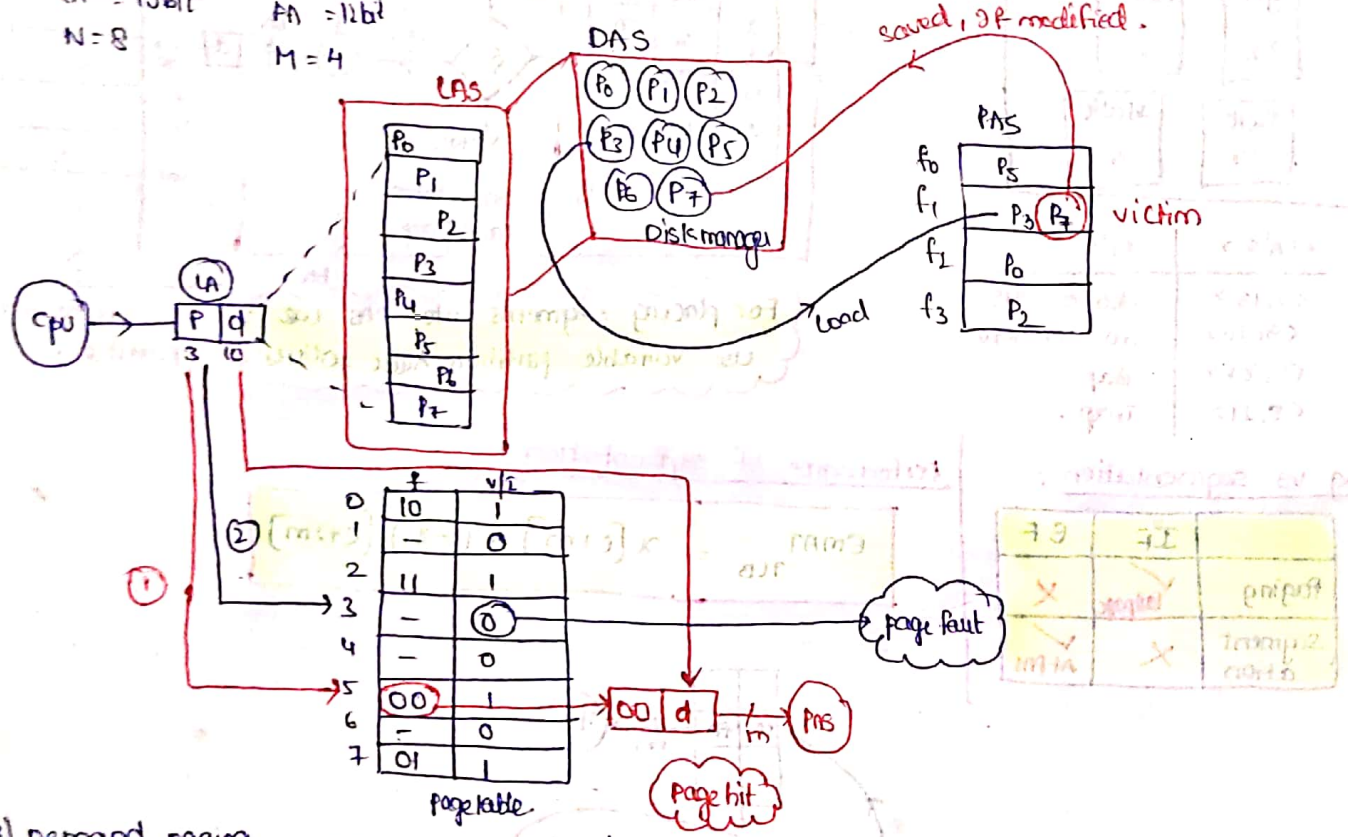
Virtual memory:

- 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 procen moved to ready Q and got scheduled and Executed further

If page fault occurs
↓
Procen 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 : <VA: 722; 715; 012; 334; 755; 854; 860; 011; 339; 964; 674; 55>
PS = 100
p: 7, d: 22

p: 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 procen 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 (working)	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

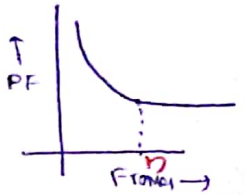
Page replacement techniques :

① FIFO :

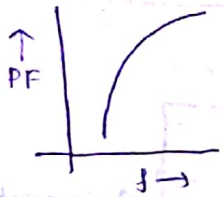
	f	v/i	TOL
0	x	1	5

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

FIFO curve



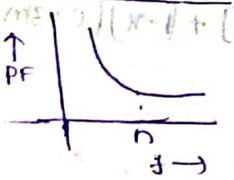
Belady Anomaly : <1 2 3 4 1 2 5 1 2 3 4 5 >



② optimal page replacement :

On 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) :

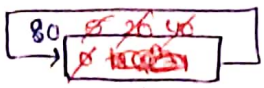
Criteria : Time of reference.

LRU Implementation :

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

Implementation

Counter method

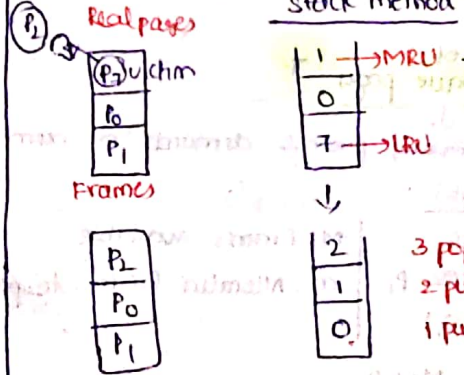


CPU cycles

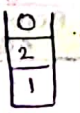
	TOR	
MRU ← P_i	~~5~~ 80	
LRU ← P_j	20	
P_k	40	
P_L	X → PF	

∴ it may fail bcoz it can count upto 2^n clocks

stack method



If no page fault also the stack should be updated

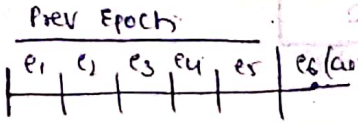


Worst case (bottom) | Best case (Top next Element)
2n operations | 4 operations

∴ More stack overhead.

LRU approximations :

① Reference bit :



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 :

	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

Current Epoch ↑

P_i 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 Anomaly

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)

Refered/not 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'

(4) 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

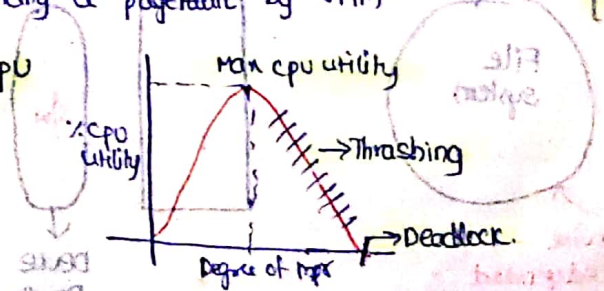
↓ Effects

- under utilization of CPU

Reasons for Thrashing :

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

Effect of servicing a pagefault by VMM



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** :

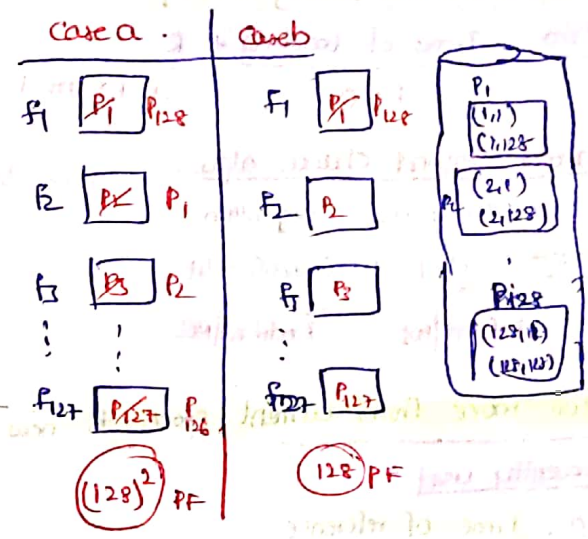
Int A [1--128, 1--128]

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

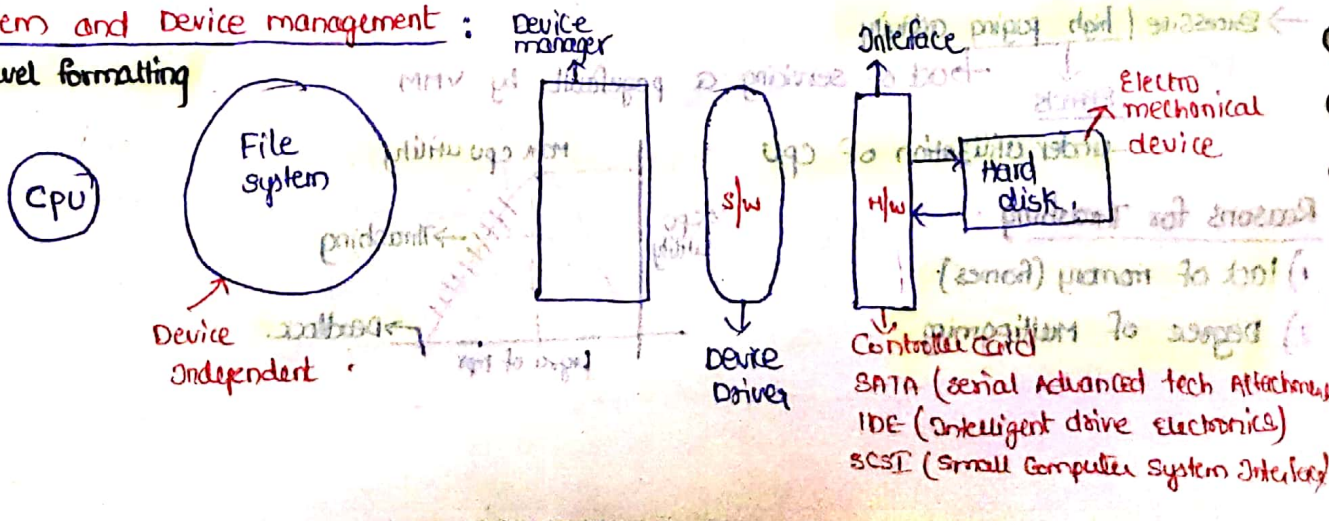
*) **Searching**

- **Linear**
  - – more thrashing
  - Less
- **Binary**
  - – more thrashing

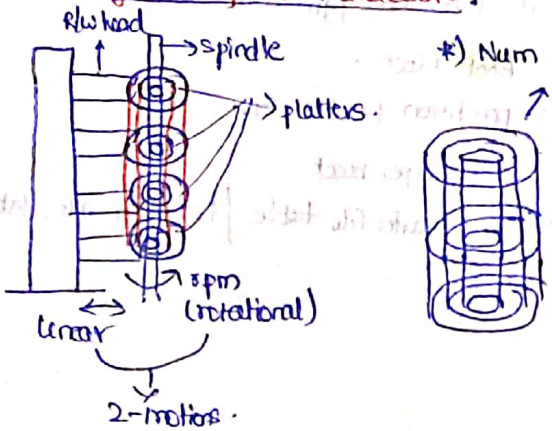
*) stack : less thrashing
Hashing : more thrashing
Pointers : more thrashing

File system and Device management :

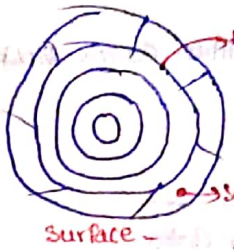
→ low level formatting

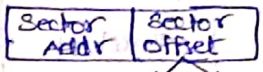


Physical geometry of Harddisk :



*) 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)

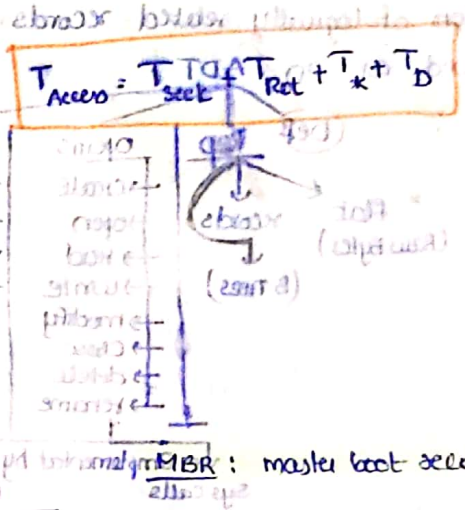
3) Transfer time = $\frac{YR}{X}$ ⇒

Track Size → 'x' B
Sector size → 'y' B
Rot time → 'R'
(Rot) x Bytes
? ← Y Bytes
(TT) $\frac{YR}{X}$

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

'X' B → R sec.
1 B → $\frac{R}{X}$ sec.

$T_{Access} = T_{Seek} + T_{Rot} + T_{*} + T_{D}$



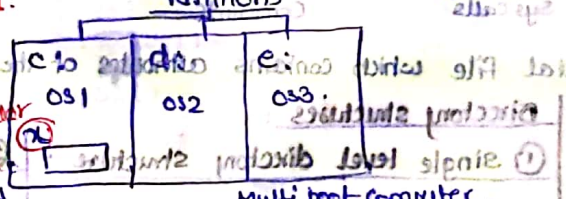
MBR : master boot record

Disk logical structure (High level formatting)

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

Floppy drives. a: b:

Partitions



Multi boot computer.

Booting process :

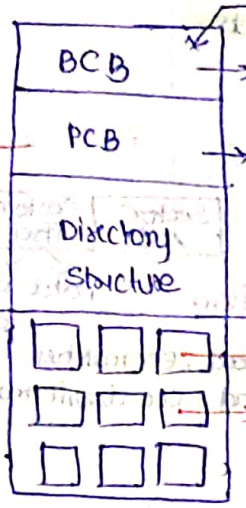
(H/w test) ← POST (Power on self test)
checks all H/w
↓
checks all I/p & o/p systems ← BIOS (Basic I/O system)
↓
Bootstrap (Bootstrap is a pgm in Rom which loads MBR to main memory & gives control to Boot loader)
↓
Boot loader (Provides option to select os)

GRUB	LILO	NT LOR
Grand unified Boot loader (UNIX)	Linux loader (Linux)	Neuro-tech Loader (Windows)

↓
Shell pgm / GUI
↓
Booting completed.

Partition structure :

Boot loader loads os from it (Bootable pgm like kernel.exe)



BCB → Boot Control Block ← UFS : Boot Block · NTFS : partition Boot sector

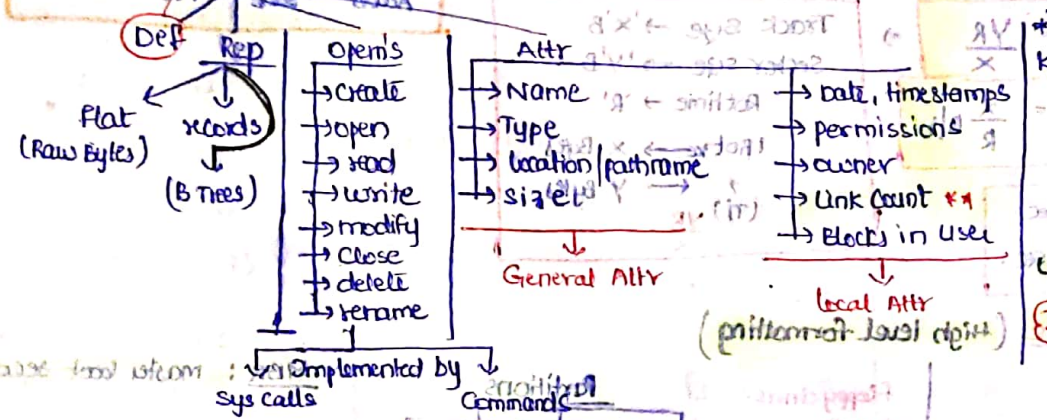
PCB → partition Control Block ← UFS : super block · NTFS : Master file table | FAT (File Alloc table)

If gone/corrupted Everything is gone.

Files & Directories :

collection of logically related records.

: viewed as an ADT T : T



*) Attributes of file are kept in FCB

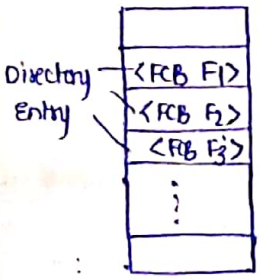
File Control Block

UNIX (UFS) → I-Node

Windows (NTFS / FAT 32) → Dir Entry

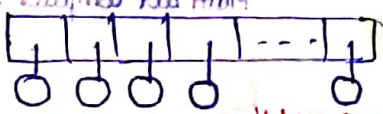
Directories : A special file which contains attributes of the file in a well organized structure.

Abstract view :



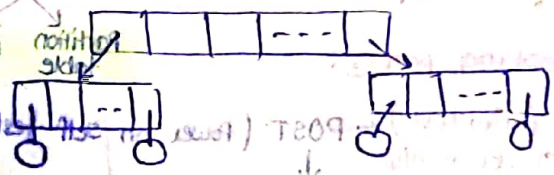
Directory structures :

① single level directory structure :

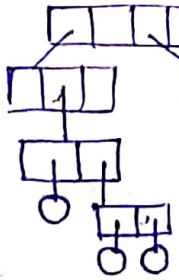


∴ More searching time, Can't have same name for the files.

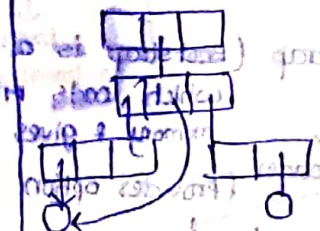
② Two level directory structure :



③ Tree structured directory :



④ Acyclic graph Directory structure :



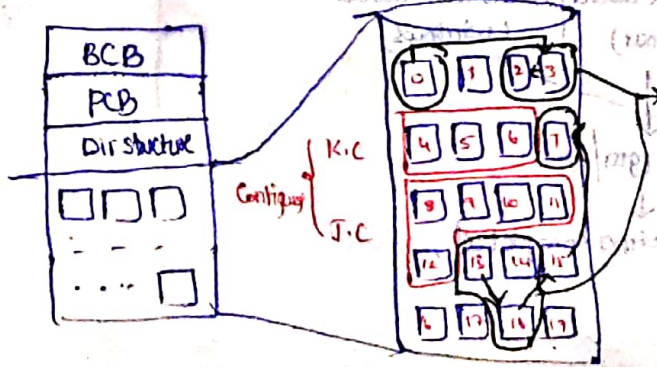
File sharing ✓

Replication → mem waste · Inconsistent

Pointing → No mem waste → Imp by Int counter

File Allocation methods :

- Contiguous Alloc
- Noncontiguous / linked Alloc
- Indexed Alloc



DBA (Disk Block Addr) ⇒ 16 bits ⇒ 2^{16} ⇒ 64K blocks

DBS (Disk Block size) ⇒ 1KB ⇒ Block size.

$2^{16} * 1KB \Rightarrow 64MB$

Contiguous Alloc :

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

Performance Issues

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

☺ Faster

☹ - Slower
- Pointer consumes disk space
- vulnerability of ptrs leads to file truncation.

Non-contiguous 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.

Indexed Allocation :

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

16) Index block

1	D } 1KB
14	D } 1KB
17	D } 1KB.

Unix Directory Implementation and Allocation method :

FN	Inode
K.C	23

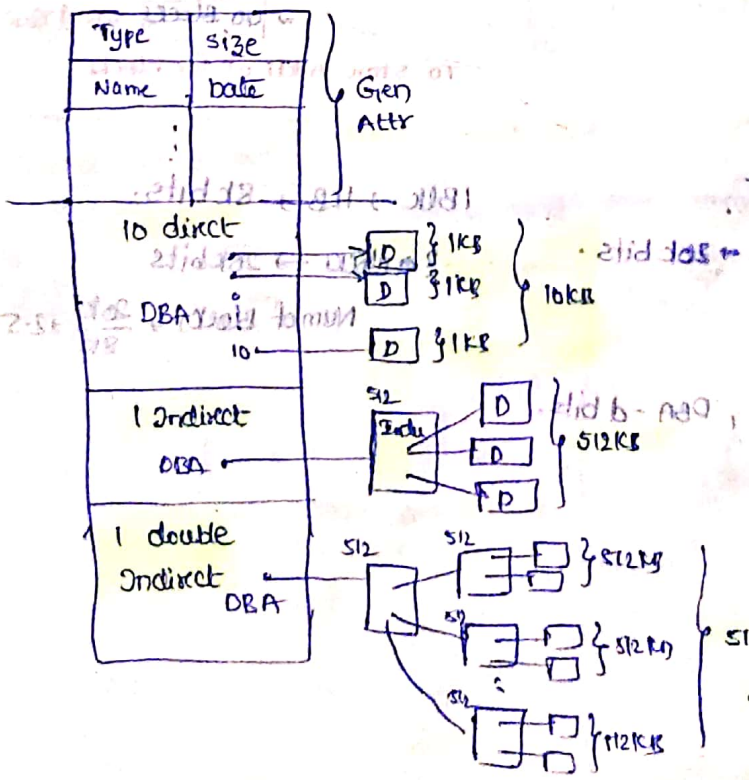
DBA : 16 bits
DBS : 1KB

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

wrt Inode

Filesize : 10KB + 512KB + 256MB
: 256.522 MB

Inode :



Dos | Windows Directory structure :

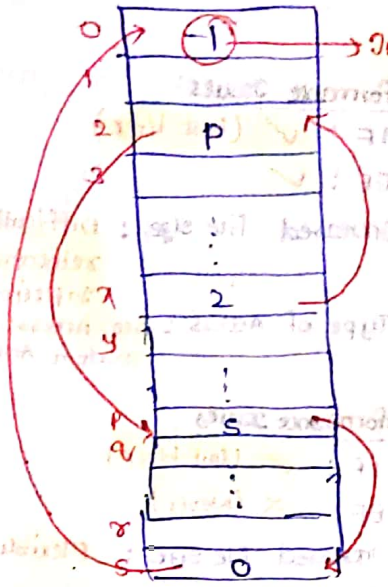
	Gen	Local
FN	- - -	First DBA
K.C	- - -	2

*) 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



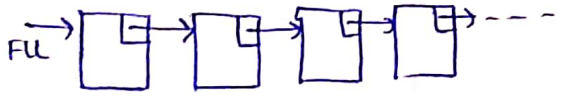
(Tabular linked Allocation)

DBS = 1KB, DBA = 16 bits, Disk Size = 20MB, Blocks = 20k.

Disk free space mgmt :

Free space Allocation techniques :

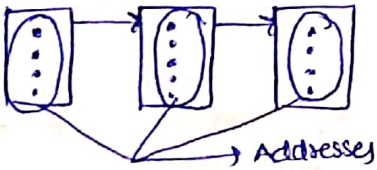
① Free linked list (FLL) :



All free blocks.

*) Linked list of free blocks.

② Free list :



Addresses of free blocks.

*) Best policy to Allocate free blocks because no searching is required

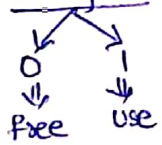
1 Blk → 1KB / 16 → 512 Addr.

20k blocks → 40 (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.

2 Blocks → 20k bits

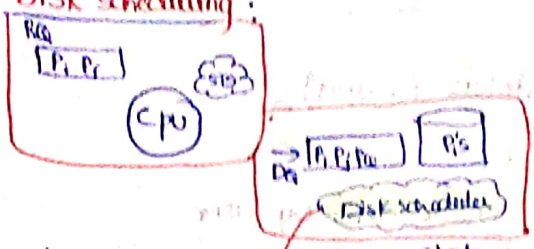
Num of Blocks → 20k / 8k → 2.5

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 sum of seek of a disk

Techniques :

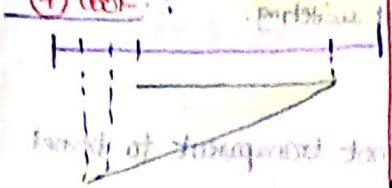
① FCFS : First come first serve

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

③ scan / elevator :

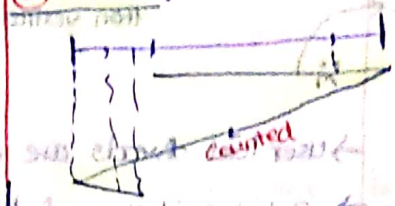
∴ starvation to process in opposite order

④ look :

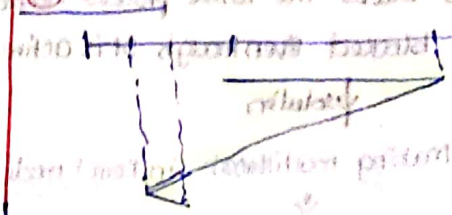


- Intelligent

⑤ C-scan :



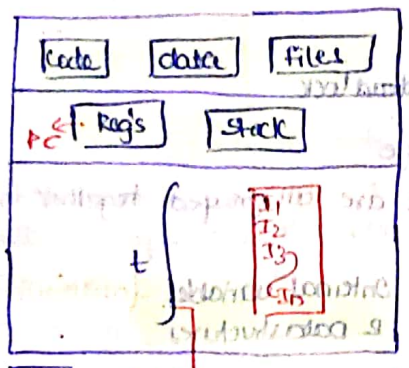
⑥ c-look



Leftovers :

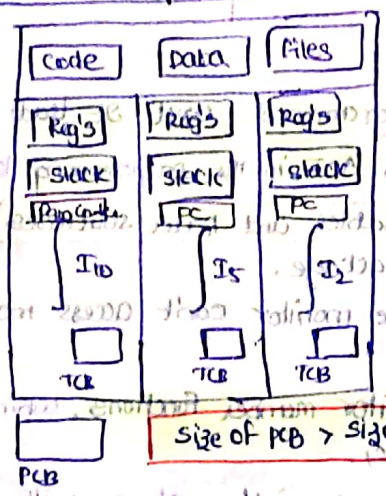
Threads & Multithreading :

Single thread process :



Indicates our pgm contains Branched Instructions .

Multithreaded process :

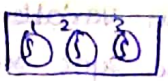


Size of PCB > Size of TCB

Thread definition :

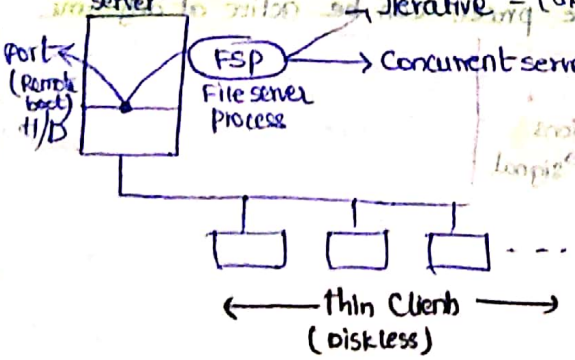
→ Light weight process
→ unit of cpu utility
→ Active Entity

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



Evolution of Threading :

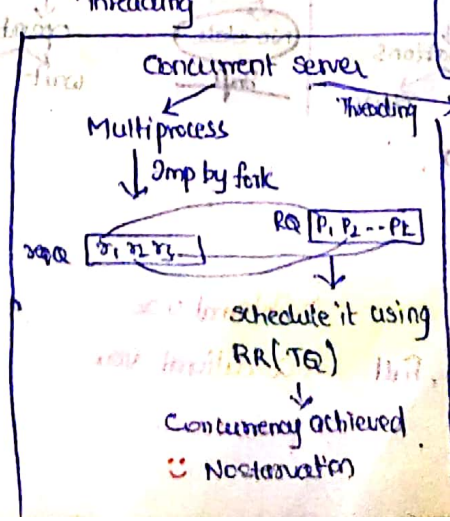
Client server Arch



*) Booting

Local boot → pc's

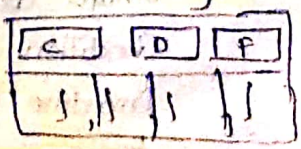
Remote boot → on server → Ex:- lab



∴ If k process should run k copies of code data files O/H created

SOL

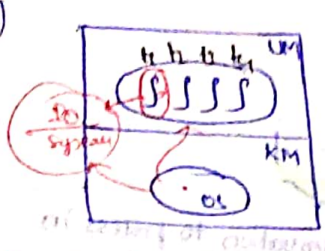
Threading



Types of threads :

1) User level threads (Java threads, posix threads)
2) Kernel level threads → (portable as interface for users)

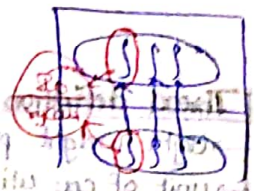
*)



In OS view the req mode by process 'p'
so, the OS blocks the total process thereby
t1 t2 t3 get blocked Eventhough it is active.

↓ solution

making multithreads in kernel mode



User level threads :

- thread registration can be done by user
- ☺ flexibility
  Transparency
  fastest context switching (bcz of no mode switching)
  (Horizontal switching rather than vertical switching.)
- ☹
  → user level threads are not transparent to kernel
  → Context switching of user level is faster than kernel level.

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 (attributes) & data structures

*) however they can active monitor member functions, which often can manipulate monitor data structures & internal variables.

*) Monitors also contain condition variables along with mutex lock

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

*) Monitors have an important property that only one process can be active at any time

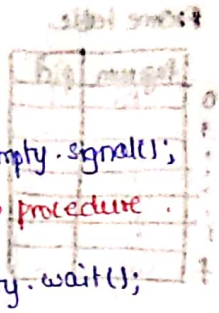
*) semaphores
  → state (value) — operations → up / down

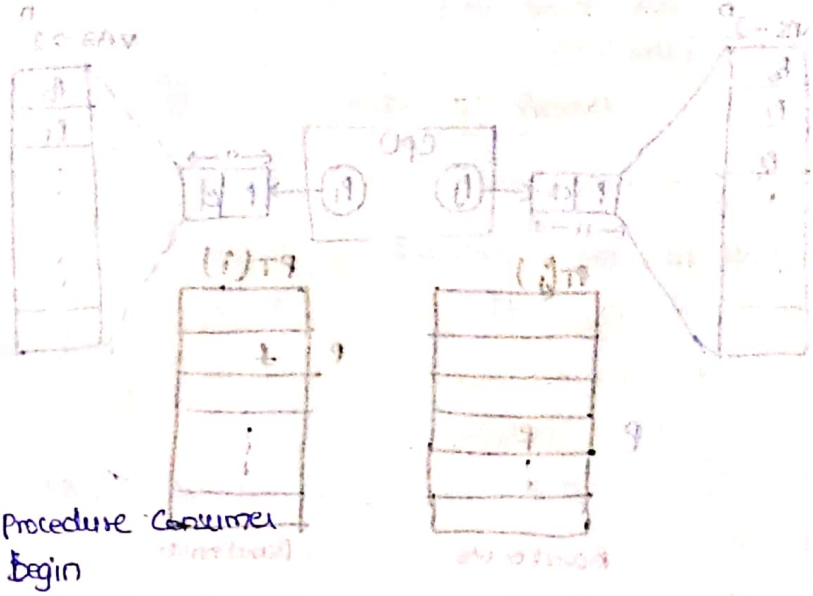
Monitors
→ no state
→ operations → wait / signal
→ 3 op

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
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.
```

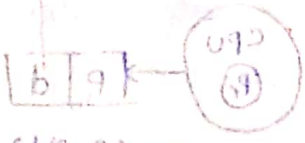




```
Procedure producer
begin
    while(True)
    {
      a) produceitem(&itemp);
      b) Prod_Cons.enter ;
    }
End.
```

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

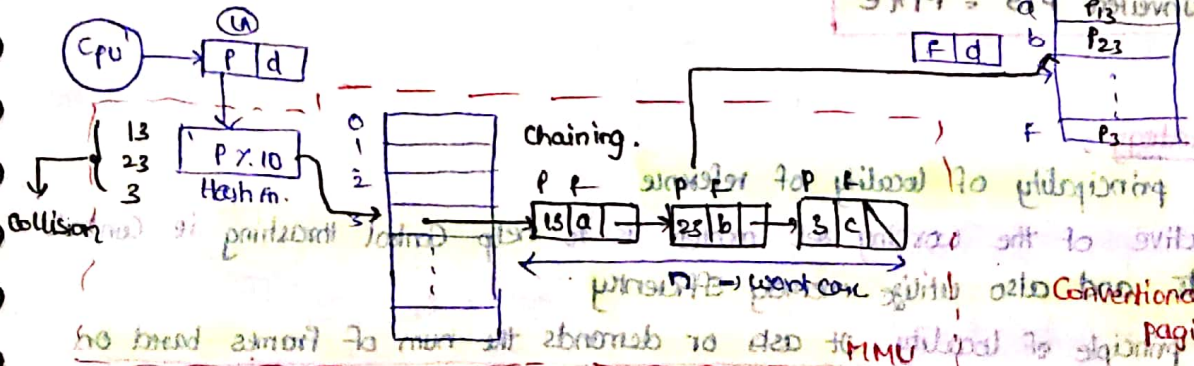
Outside fn wants to access monitor so Implicitly mutex lock is applied so that one process can run on monitor.



Hashed paging :

To reduce page table size

- Multilevel paging
- Hashed paging
- Inverted paging



Conventional paging : O(1)

☺ Space Efficient

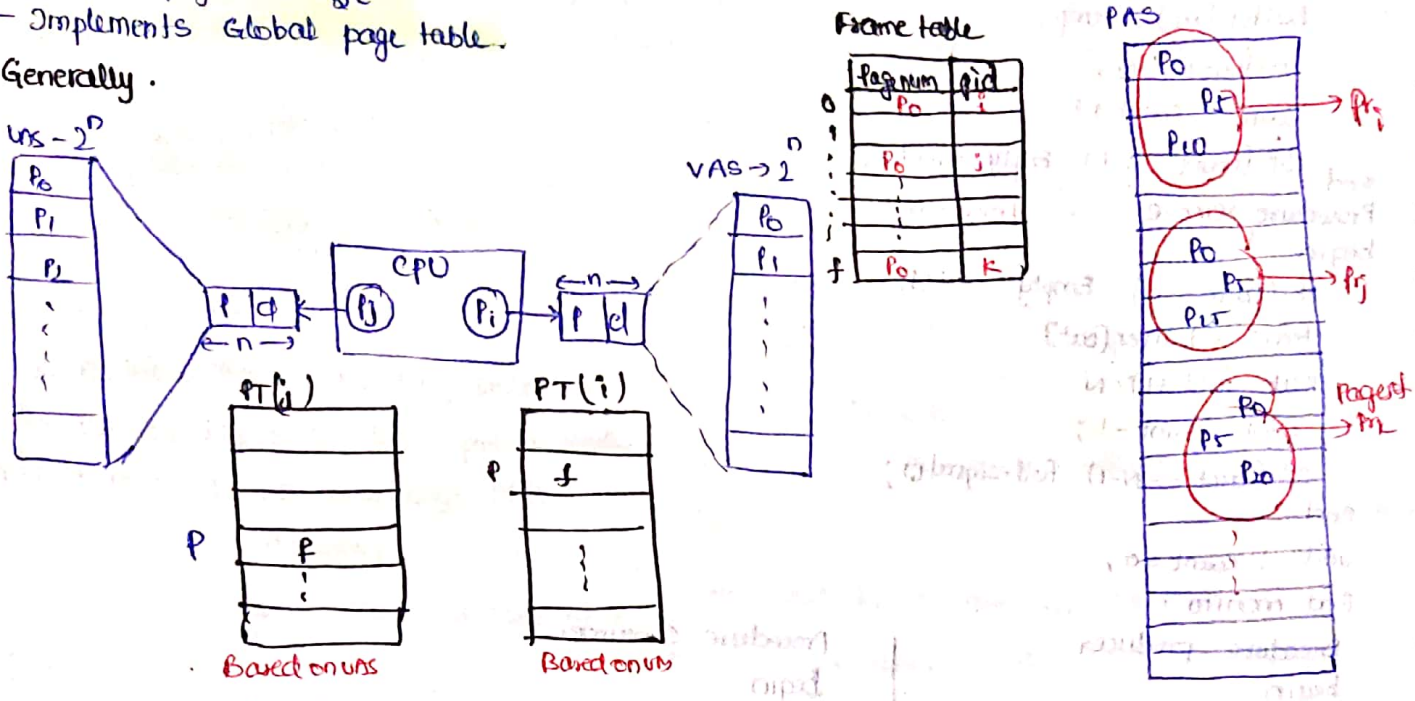
☹ O(n) → Sequential Access.

Inverted paging:

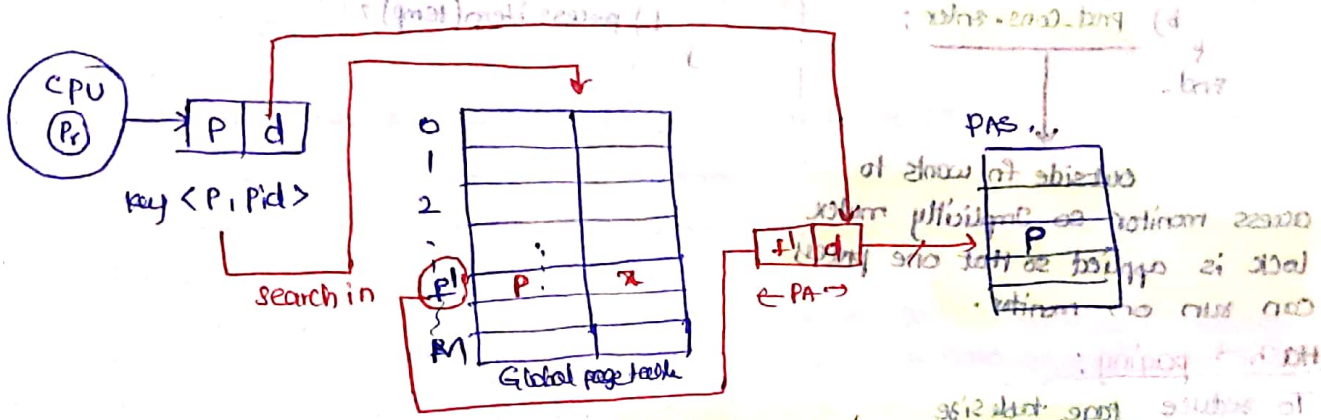
→ reduce page table size

– 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 Control 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

```
10KB main()
     {
       f();
     }
5KB  f()
     {
       g();
     }
```

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

Prog size : 55 KB
Let PS : 1KB

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 P_{rk} : 7, 0, 1, 2, 0, 1, 7, 6, 7, 0, 1, 3, 15, 18, 17, 16, 15, 33, 35, 38, 39, 42, 40, 33, 34, 35, 32

Δ = 10

Now,

Working set window = WSW_k^i = { set of unique pages refered by process k from time t_i during past Δ references }

Here

Δ → Guess value

let Δ = 10

WSW_k^i (Δ=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

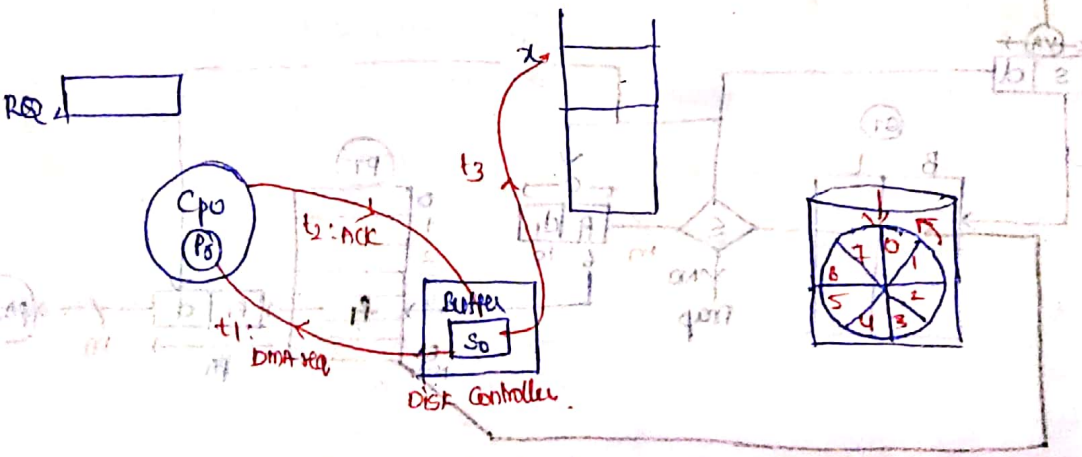
$D = \sum_{i=1}^{k} WSWS_i$, M = Frames Avail

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 c/u

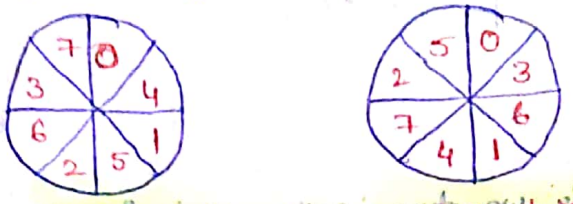
↓

At this time the disk read some more sector and it is lost

Solution for the problems is..

H/w sol
Providing max buffers
:) Costlier

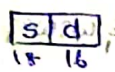
S/w solution
↓
Interleaving
↓
Single Interleaving | double Interleaving



Segmented paging:

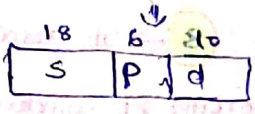
→ paging is used in segmentation to overcome the problem of external fragmentation in PAS
→ Segment is divided into pages and pages are stored in frames of memory which later are accessed through page table.
ie
Every segment is refered through page table and page table is accessed through Segment table

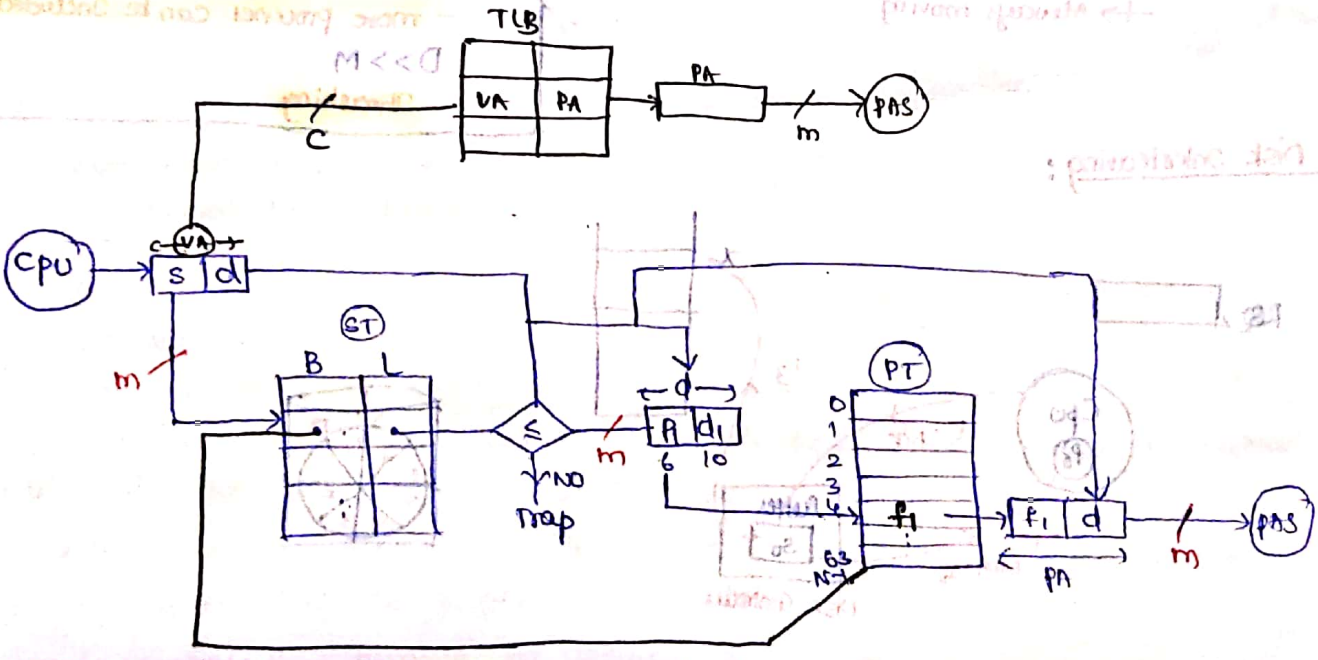
Ex:- VA : 34 bits <18, 16> max Num of seg = 2^{18}, max seg size = 2^{16} = 64KB



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

ie





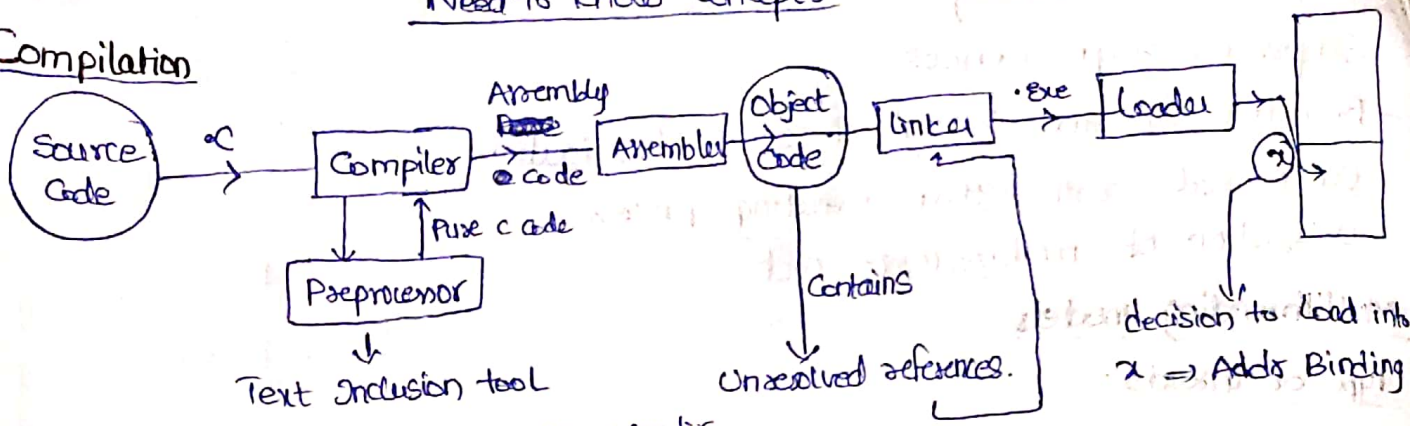
*) EMAT without TLB = 3m

EMAT with TLB = $x[c+m] + (1-x)[c+2m]$

x — The End — x

Need to know Concepts

Compilation



① File Inclusion <> → sys dir
" " → curr dir

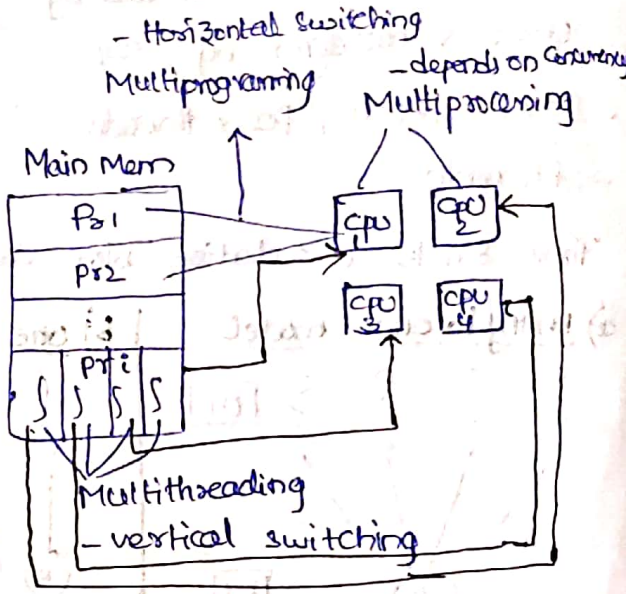
② Macro Expansion # define N 5

③ Remove Comments.

Threads :

- Pgm under Execution is process
- Thread is a basic unit of cpu utilization
- Thread Comprises of Unique & shares

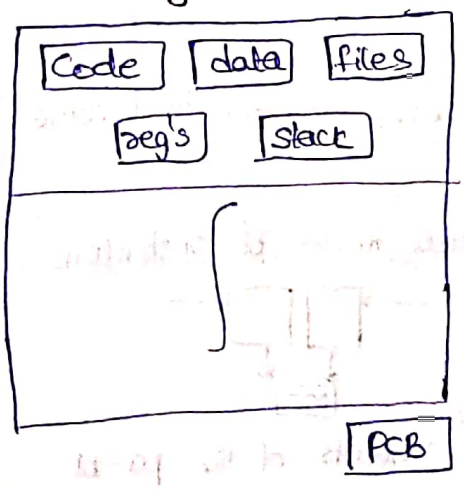
Unique	shares
(i) Thread id	(i) Code
(ii) stack	(ii) Data
(iii) reg's	(iv) resources like files signals
(iv) program Counter	



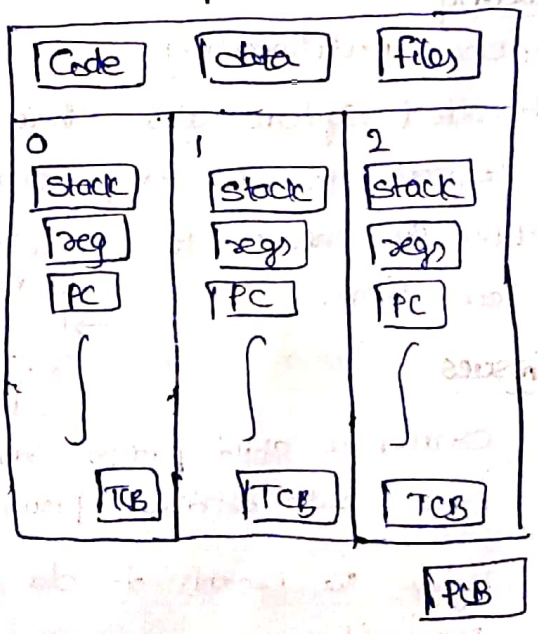
Multithreading :

- If a procen has multiple threads of control then it can perform more than one task at a time is called multithreading

single thread



Multiple threads



Benefits of multithreading:

- Improves responsiveness
- Resource sharing through thread synchronization
- Economical rather than creating process
- Utilization of multiprocessor Arch

Multithreading models

Type of threads:

① User threads:

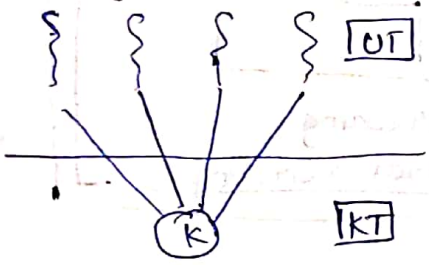
- Threads that are created by user & managed by user
- Java threads, posix threads
- flexible

② Kernel threads:

- Threads that are created & managed by OS.
- Limited threads -

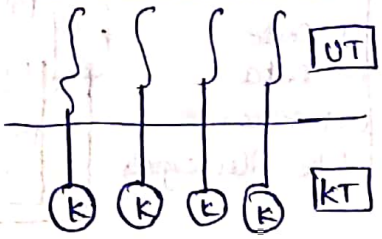
There Exists a relation b/w user & Kernel threads is called multithreading model.

a) many to one model



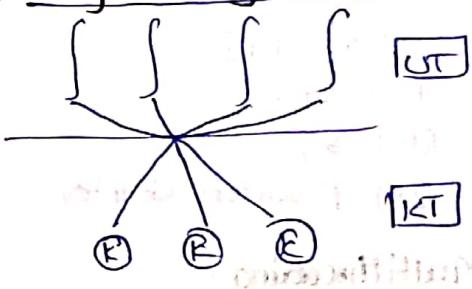
∴ If one thread is blocked Every thread is blocked

b) one to one model



∴ As Kernel threads are Limited it effects flexibility of user level threads

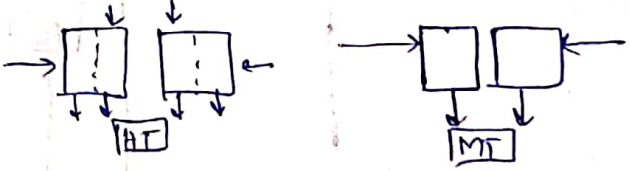
c) many to many model



- Practical model
- provides flexibility of UT
- Thread scheduling

Hyperthreading:

- Simultaneous multithreading
- Hyper threaded Systems allows their processor Cores resources to become multiple logical processors for performance.
- It Enables the processor to Execute two threads or set of instructions at the same time.



Threading issues:

① fork() : Creates a child process containing the contents of the parent process with different process id.

② Exec() : replace the contents of old process with new process with same process id.

Issue 1:

If one thread in a program calls fork(), does the new processor duplicate all threads or the new processor is single threaded?

Sol: UNIX systems have chosen 2 versions of fork()

① fork_all() - duplicates all threads

② fork_current() - duplicates current thread.

When to use:

Pgm

fork() --

==

→ fork_current

Exec()

End

Pgm:

fork()

≡≡≡

→ fork_all()

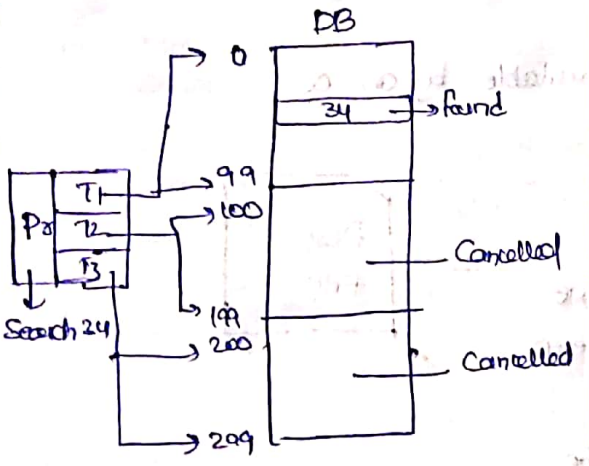
End

Issue 2: Thread cancellation

↓

task of terminating a thread before it has completed

Consider a scenario searching for 34



Cancellation of a target thread may occur in two different scenarios

① Asynchronous cancellation:

One thread immediately terminates the target thread.

② Defered cancellation:

The target thread periodically checks whether it should terminate allowing it an opportunity to terminate itself in an orderly fashion.

Problem with Thread cancellation:

- Resources have been allocated to a cancelled thread
- When a thread is cancelled, threads hold resources and gets terminated making that resource unavailable

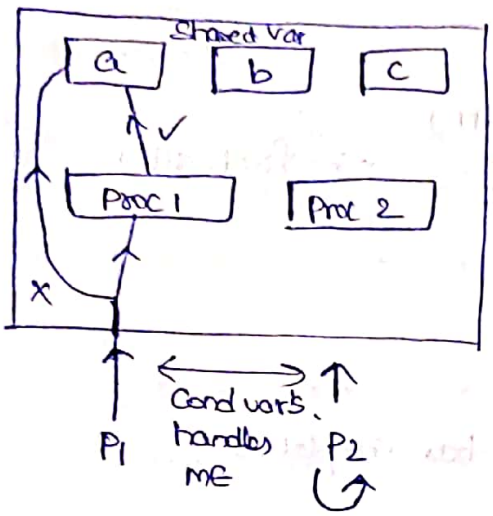
Asynchronous

↓sol

- OS handles to reclaim system resources. ∵ burden to OS.
- Defered cancellation ✓

Monitors :

- Solves the problem of CS
- Deadlock free
- Always Ensure ME
- Semaphores → State, Value
- Monitors → State x, Value.

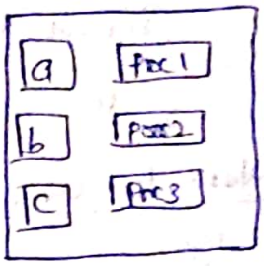


Monitor can be viewed as ADT

Def : A monitor is a module that contains
- shared data
- procedures that operate on the shared data.

Rep: syntax of monitor

```
Monitor {
    condition var;
    variables;
    Procedure1() {
    }
    Procedure2() {
    }
};
```



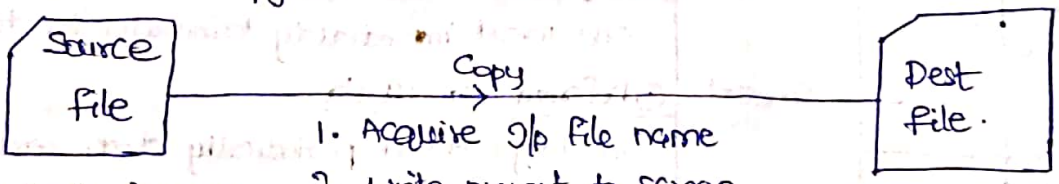
Opers : wait, signal

Attributes : var, procedure,

System calls : (Routines written in C & C++)

- It provides an interface to the service made available by an OS.
- Invoked from user pgm's.

Ex:-



System call sequence :

1. Acquire i/p file name
2. Write prompt to screen
3. Accept input
4. Acquire o/p file name
5. Write prompt to screen
6. Accept input
7. open input file
8. If file doesn't exist then ABORT
9. create output file
10. If file already exist then ABORT
11. Read from input file char wise.] loop
12. Write to outputfile.] until EOF
13. Close output file
14. Write completion message to screen
15. Terminate normally

Types of system calls

- **Process Control**
  - End, abort
  - Load, Execute
  - wait
  - signal
  - allocate
  - deallocate
  - create
  - terminate
  - get process attr
  - set process attr

  PCB
- **File manipulation**
  - Create
  - delete
  - open
  - close
  - read
  - write
  - reposition
  - get file attr
  - set file attr

  FCB
- **Device mgmt**
  - req device
  - release device
  - read
  - write
  - reposition
  - logically attach
  - logically detach
  - get dev attr
  - set dev attr

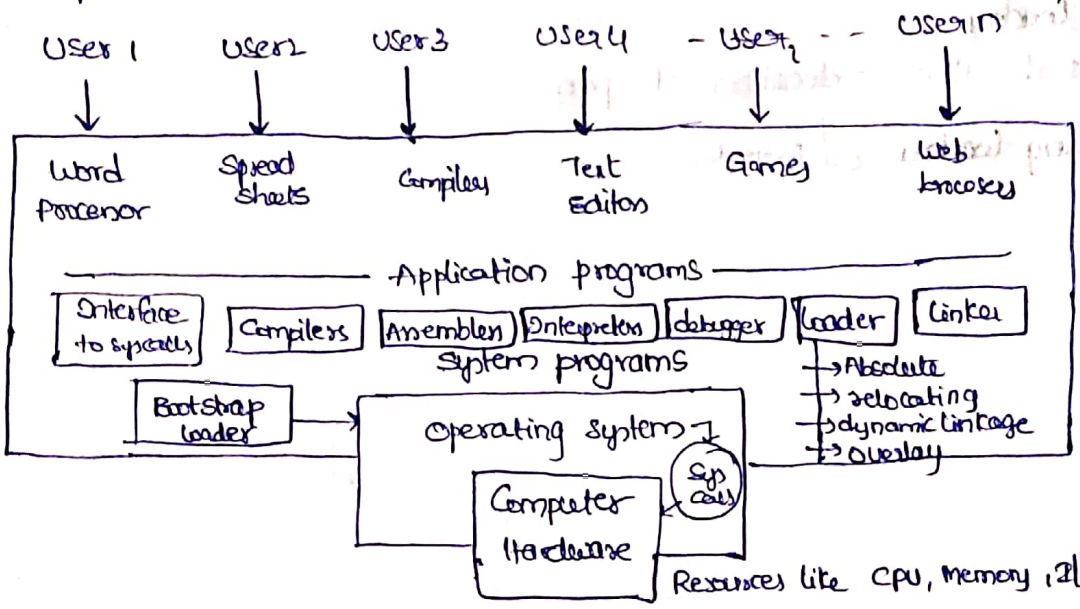
  FCB
- **Info maintenance**

get	set
time	time
date	date
Procn	Process
File	File
device	device

  attr
  - get sys data
  - set sys data.
- **Communications**
  - Create Comm
  - delete Comm
  - send msg
  - recieve msg
  - attach rem dev
  - detach rem dev
  - transfer status information.

System programs :

An important aspect of a modern system is the collection of sys programs.



- System programs provide a convinient environment for pgm dev & Execution.
- Some of them are simply user interface to sys calls while others are considerably complex.

System programs

- **File mgmt**
  - Deals with files
- **status info**
  - provides info abt Procn, device, files.
- **Communication**
- **file modification**
  - modifies files,
- **Programming lang support**
  - Compilers
  - Interpreters
  - Debuggers
  - Assemblers
  - Loaders
  - Linkers

Types of loaders

↓

Loads the procen into memory

Functions of loader : Allocation of memory
linking
relocation
loading

Types :

① **Compile & go loader :**

- The Assembler itself places the assembled instructions directly to the designated memory locations for Execution.

② **Absolute loader :**

- It loads into the location prescribed by the assembler
  Ex :- bootstrap loader.

③ **relocating loader**

- loaders that allow relocation of pgm.

④ Direct linking loader ↙ Example.