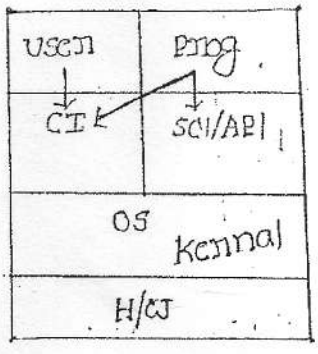
command Interpreter:

* A program that interprets the commands of the user.
* command interpreter acts as interface between program and Kernal

command interpreter

API: Application prog interface → WIN

SCI: system call interface → UNIX



NOTE: * User can interact with OS through command interpreter it translates the user inputs/requests into SCI/API.

2. OS is visualized as Resource Manager.

Resources

Hardware
* CPU
* Memory
* I/O
* Registers
→ Directly under the control of the OS

Software
* Device drivers
* semaphores
* Msg
* Monitors.
→ integrated part of the OS

Functions & Goals:

Functions

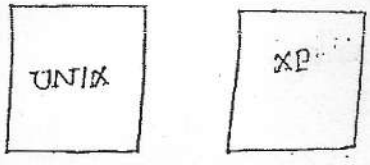
=> Resource Management
=> Memory Managment
=> Process Managment
=> File Managment

Goals:

=> convenience (userfriendly)
=> Efficiency (Resource utility)
=> Portability (diff platform)
=> Reliability
=> scalability

NOTE: Primary Goal: convenience
Secondary Goal: Efficiency.

Eg:

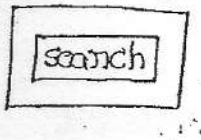


* To find a particular content in "UNIX" OS we must write its syntax

Eg: $find
%.name, *text.c*

so that it requires exact syntax, but it takes less time to execute the given statement.

* To find the same content in "XP" O.S it is an easy process that any user can approach through it.

Eg: 

find.exe.

it takes more time than UNIX to execute, but it is more familiar to all kinds of users to operate.

NOTE: * "convenience" is more important than "efficiency".
* some real time systems "Efficiency" is more important

Functions & Goals:

Functions

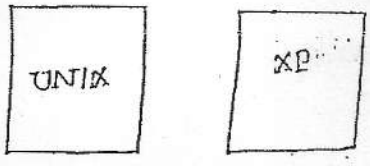
=> Resource Management
=> Memory Managment
=> Process Managment
=> File Managment

Goals:

=> convenience (user friendly)
=> Efficiency (Resource utility)
=> Portability (diff platform)
=> Reliability
=> scalability

NOTE: Primary Goal: convenience
Secondary Goal: Efficiency.

Eg:

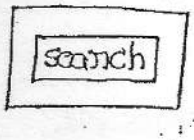


* To find a particular content in "UNIX" OS we must write its syntax

Eg: $find
% name, *text.c*

so that it requires exact syntax, but it takes less time to execute the given statement.

* To find the same content in "XP" O.S it is an easy process that any user can approach through it.

Eg: 

find.exe.

it takes more time than UNIX to execute, but it is more familiar to all kinds of users to operate.

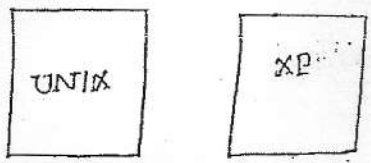
NOTE: * "convenience" is more important than "efficiency".
* some real time systems "Efficiency" is more important

Functions & Goals:

Functions	Goals:
=> Resource Management	=> convenience (user-friendly)
=> Memory Managment	=> Efficiency (Resource utility)
=> Process Managment	=> Portability (diff platform)
=> File Managment	=> Reliability
	=> scalability

NOTE: Primary Goal: convenience
Secondary Goal: Efficiency.

Eg:

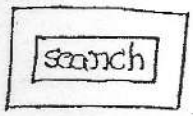


* To find a particular content in "UNIX" OS we must write its syntax

Eg: $find
% name, *text.c*

so that it requires exact syntax, but it takes less time to execute the given statement.

* To find the same content in "XP" O.S it is an easy process that any user can approach through it.

Eg: 

find.exe.

it takes more time than UNIX to execute, but it is more familiar to all kinds of users to operate.

NOTE: * "convenience" is more important than "efficiency".
* some real time systems "Efficiency" is more important [illegible] "convenience"

Evolution of OS:

* First Generation 1940's : Card Readers/Punch cards : No OS.
* Second Generation 1970's : ⇓ interact manually with H/w.

  Batch processing : No OS. ⇒ Magnetic Tapes
  ⇓ (Spooling)
* Third Generation 1980's : Disk Technology : OS was introduced.

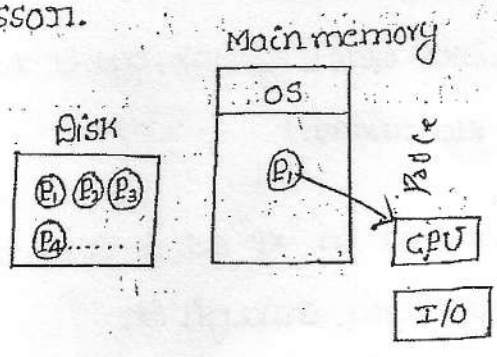
Disk Technology

- Uniprogram
- Multiprogram
  - single user Ex: DOS
  - Multi user Eg: WIN.

* Uniprogram : Execution of programs serially.
* Multiprogram: Multiple programs can be executed by the Process parallely.

Uniprogram:

once a Job can be selected by processor it can not be forced to leave the CPU until it is finished, and main memory can store only one program at a time, which can be run by the processor.



DisAdv:

* Multitasking is not Possible.
* CPU utilization is poor i.e idleness of CPU.

Multiprogram:

In main memory multiple programs will be stored at any given point of time, while one program is being Processed other waits in queue, once a process is either

Evolution of OS:

* First Generation 1940's : Card Readers/Punch cards : No OS.
* Second Generation 1970's : ⇓ interact manully with H/w.

Batch processing : No OS. ⇒ Magnetic Tapes

(Spooling)

* Third Generation 1980's : Disk Technology : OS was introduced.

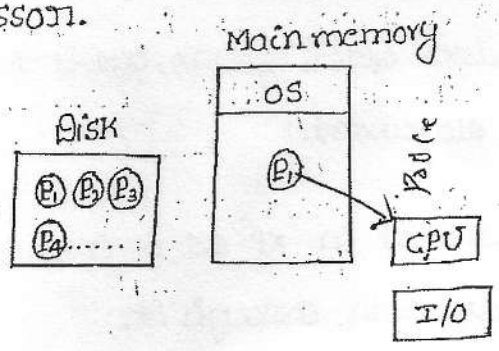
Disk Technology

- Uniprogram
- Multiprogram
  - single user Ex: DOS
  - Multi user Eg: WIN.

* Uniprogram : Execution of programs serially.
* Multiprogram : Multiple programs can be executed by the Process parallely.

Uniprogram:

once a Job can be selected by processor it can not be forced to leave the CPU until it is finished, and main memory can store only one program at a time, which can be run by the processor.



DisAdv:

* Multitasking is not Possible.
* CPU utilization is poor i.e idleness of CPU.

Multiprogram :

In main memory multiple programs will be stored at any given point of time, while one program is being Processed, other waits in queue, once a process is either

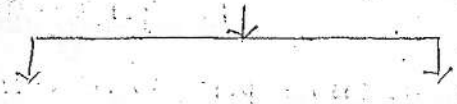
completed on goes for I/O on preempted, the processor picks up some other program to process.

Adv:

* It enables multitasking.
* CPU utilization is high (Maximum).
* Multiplexing of CPU among ready programs in memory.

=

Multiprogramming



Non-preemptive	Preemptive
→ Voluntary release	→ Forcefully
→ completion	→ complete
→ I/O, system call	→ I/O
→ Event services.	→ Timer, priority.

Non-preemptive:

Ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

* Non-preemptive OS : windows 3.0 (first version)
  DOS. 3.1
  3.11

DisAdv:

* starvation
* priority not possible.

Preemptive:

Running process can be forced to released based on

→ complete
→ I/O
→ Timer, priority.

completed on goes for I/O on preempted, the processor picks up some other program to process.

Adv:

* it enables multitasking.
* CPU utilization is high (Maximum).
* Multiplexing of CPU among ready programs in memory.

=

Multiprogramming

Non-preemptive
- → voluntary release
- → completion
- → I/O, system call
- → Event services.

Preemptive
- → forcefully
- → complete
- → I/O
- → Timer, priority.

Non-preemptive:

Ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

* Non-preemptive OS : windows 3.0 (first version)
  DOS. 3.1
  3.11

DisAdv:

* starvation
* Priority not possible.

Preemptive:

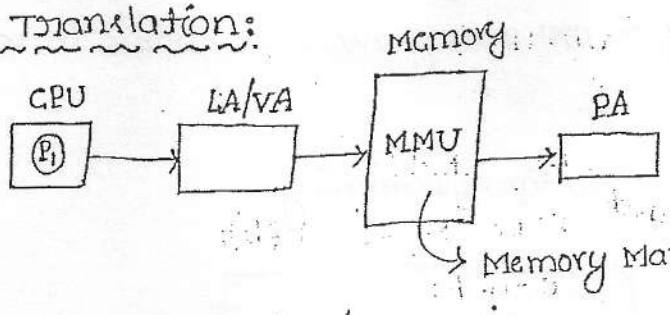
Running Process can be forced to released based on
- → complete
- → I/O
- → Timer, priority

Architecture support for Multiprogramming:

* **DMA:** (Direct Memory Access)

Process can directly access the memory for I/O operation without processor invervention with the help of DMA control

* **Address Translation:**



CPU issues a virtual/logical memory address. MMU translates it into actual physical address of the MM.

* **2-Mode operation:**

CPU has 2 different modes to run the processes

User Mode	⇒ non-privileged/pre-emptive
Kernal Mode	⇒ system/supervisory/priviliged/protected non-prumptive.

Difference between UM and KM:

* UM: User prog, all application programs (Audio, video, graphics)
  KM: Process/service routine, OS services (scheduler, dispatcher memory manage..)

* UM: The Process have non-atomic execution (without preempt)
  KM: The Process have atomic execution (without Preemption)

* Program status word (PSW) (or) Processor status word:

A bit in PSW reg commonly referred as the command bit/mode bit indicates whether it is running on UM or KM

PSW:

Mode bit

0 : User mode
1 : Kernal mode.

* Mode shifting:

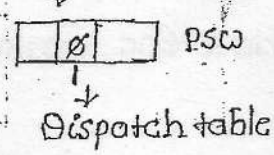
```
main()
{
User mode { int a,b,c;
            scanf("%d%d",&a,&b);
            c=a+b;
Kernal  => fork() //=> system call (SVC), create child proces
UM  mode => printf("%d",c); => BSA
}
```

Q/B

NOTE: System calls are usually invoked by using s/w interrupts.

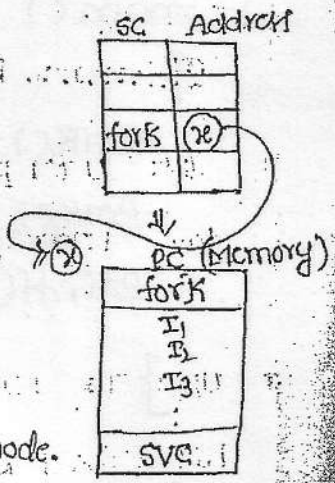
fork() → SVC ——→ s/w interrupt → ISR

(Priviliged instruction)



printf("Hello") → BSA

(Non priviliged)



* fork() creates child process and the execution of fork() is in Kernal mode because it is a part of OS and conversion takes place from UM mode to system mode.

* Execution of SVC involves the s/w interrupts which is handled by (Interrupt service Routine) ISR

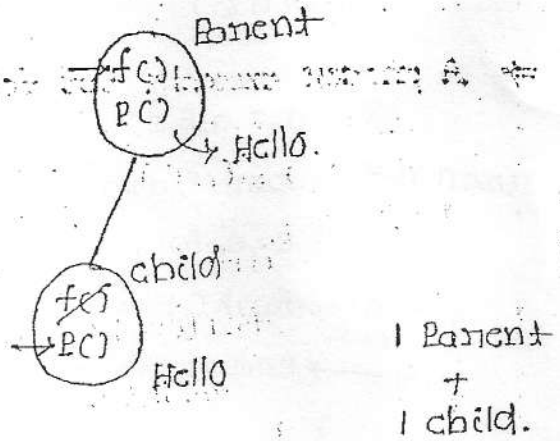
* ISR change the mode bit in PSW then it is 0 to 1. after completion of all the operations in KM again SVC will

where ISR value is changed from 1 to 0.

* Interrupt handler controls a table called "dispatch table" Dispatch table has all the entries for every system call.

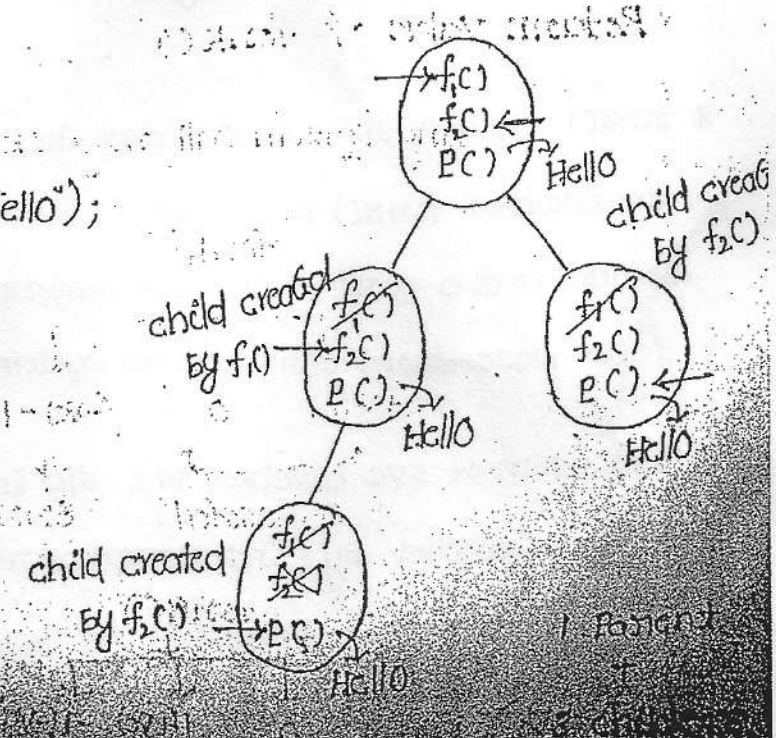
fork() implementation:

①
```
main()
{
  fork();
  Printf("Hello");
}
```



A Parent process calls fork(), it creates a child process with the same code, i.e replace of code itself, when the child process executes, it starts from the next instruction after the fork(). i.e to avoide infinite looping.

②
```
main()
{
  fork1();
  fork2()
  Printf("Hello");
}
```



* # fork() — total proces — child proces

1	2	
2	4	3
3	8	7
:	:	
n	n	
	2	2 − 1

Q 28.

⇒ A proces execute the following code.

```
main()
{
 int i, n;
 for (i=1; i<=n; i++)
    fork();
}
```

The no.of new processes created is:

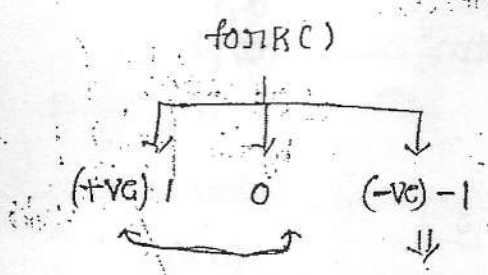
A. n B. $\frac{n(n+1)}{2}$ C. $2^n - 1$ D. $3^n - 1$

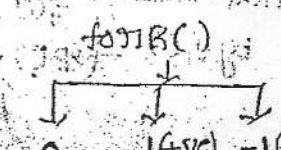
NOTE: If there are 'n' fork() calls total will be 2^n process

No.of child proces created = $2^n - 1$

Return value of fork()

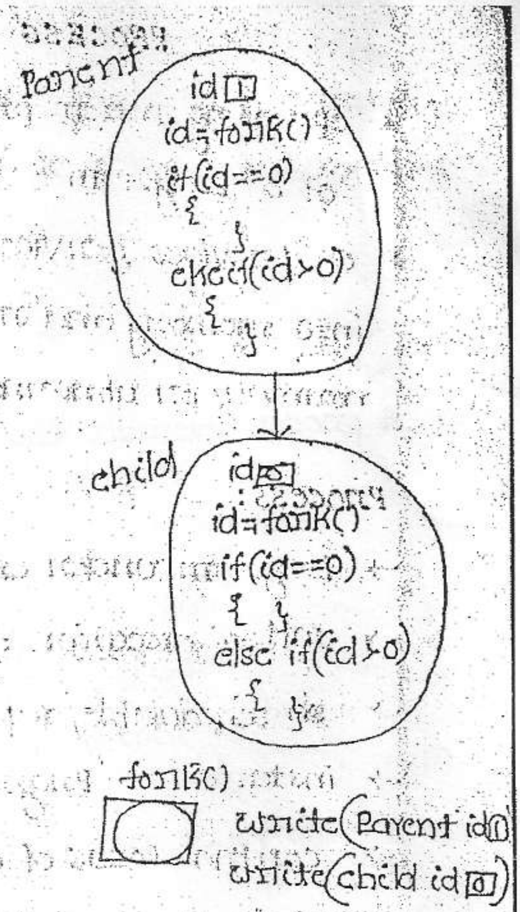
Any system call may return either +ve, 0, -1 value





⇒ Due to some problems if unable to create a new

```
main()
{
    int id;
    id = fork();
    if (id==0) // child
    {
        //child//
    }
    else if (id>0)
    {
        //Parent//
    }
    else
        Printf("Error in creation");
}
```



Parent and child address are not same.

consider the following code:

```
main()
{
    int a;            [10] 2000
    if (fork()==0)
    {
        a=a+5;        15 [10] 1000
        Printf("%d,%d\n", a,&a);   (X,Y)
    }
    else
    {
        a=a+5         15 [10]
        Printf("%d,%d\n" a,&a);   (U,V)
    }
}
```

A. U = X+10 , V = Y

B. U = X+10 V ≠ Y

C. U+10 = X V = Y

(D) U+10 = X V ≠ Y.

Let U,V be the values printed by the parent process and X,Y be the values printed by the child process. which one of above are TRUE.

PROCESS MANAGEMENT - I

* operating system provides an environment for the execution of a program
* os provides services for program execution ie load a prog into memory and run it, and end the prog execution either normally or abnormally.

Process:

→ Program under execution : Resource utilization of a sys.
→ unit of execution : (cpu)
→ sh schedulable/dispatchable unit : (thread, also Agent)
→ instance of Program : (one-to-one, one-to-many)
→ central locus of control (OS)
→ Animated spirit : (Active soul)
→ Active entity.

Program:

Program is a definite and finite set of sequence of instructions on operation and data to solve a problem.

→ MUL
→ DIV
→ ADD

Memory loc.

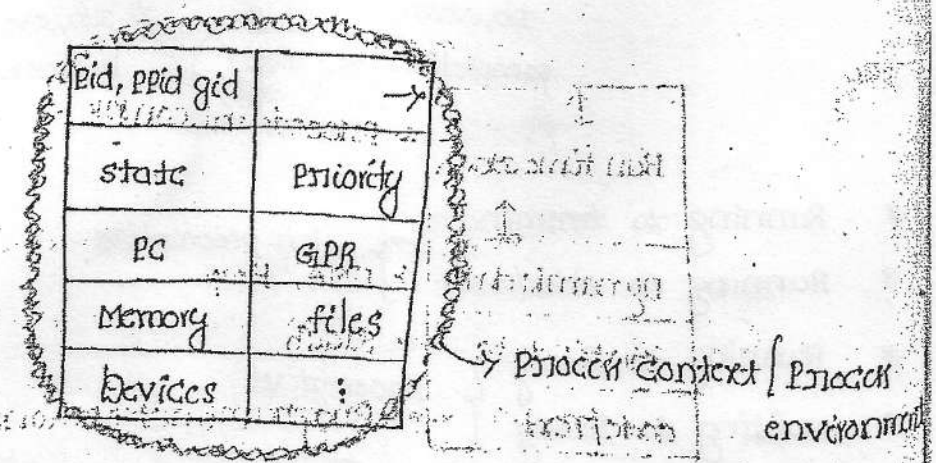
Program	Process.
* In secondary memory (disk)	* In main memory (RAM, MM..)
* Passive entity	* Active entity.
* without resource except storage.	* with resource (CPU, MM, IO
* contains → Prog code → Data → Process area	* contains → Process block

operations :

* create process
* schedule process (CPU)
* Block Process
* Run Process
* suspended process
* Resume process
* Terminate process

Attributes:

* Identification : Pid, Ppid, gid.
* CPU related attributes : Priority
  Process state
  Program counter
  Time stamp.
* Memory information : size (byte), LOC, MMU.
* File related attributes
* Device related attributes.
* Protection.



PCB : Process control Block (or), Process descriptor.
(or) Id card of the process.

TCB : Task control Block.

* PCB is a repository of information (it may vary from Process to Process)

Relationship between process and program:

Process : Program

1 : 1 → sequential execution

1 : n → concurrent execution.

Process State:

As a process executes, it changes state. The state of a process is defined in parts by the current activity of that process.

New: Process is being created.

Running: instruction are being executed.

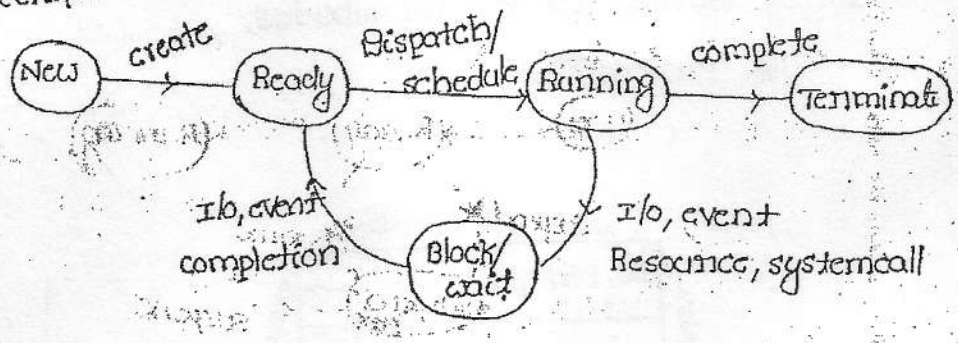
Waiting: process is waiting for some event occurs.

Ready: Process is waiting to be assigned to a processor.

Terminate: The process has finished the execution.

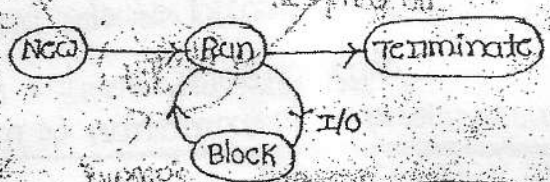
State Diagram of a process:

Non-preemptive:

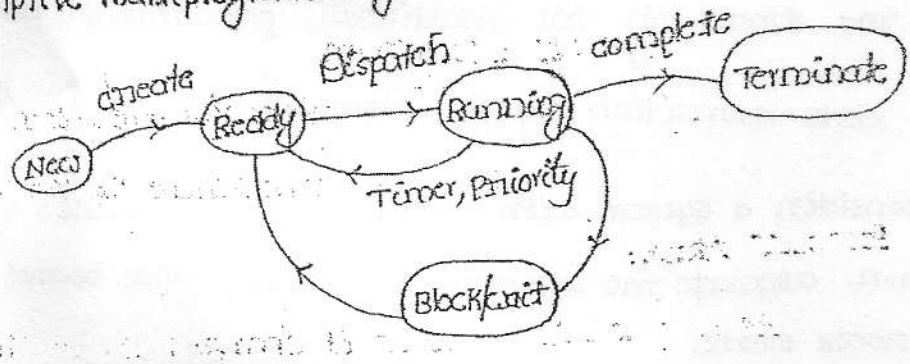


* Running to terminate } Non-preemptive
* Running to Block/wait }
* Running to Ready } Preemptive.
* waiting to Ready }

State dig for uniprogramming: No - Ready is present.

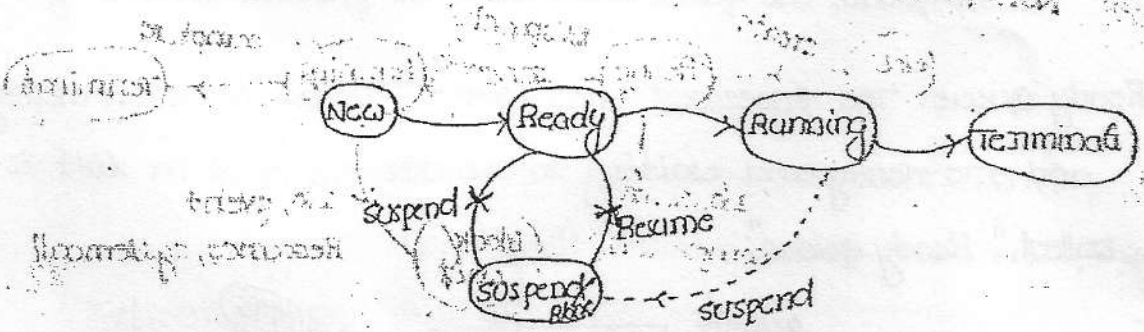


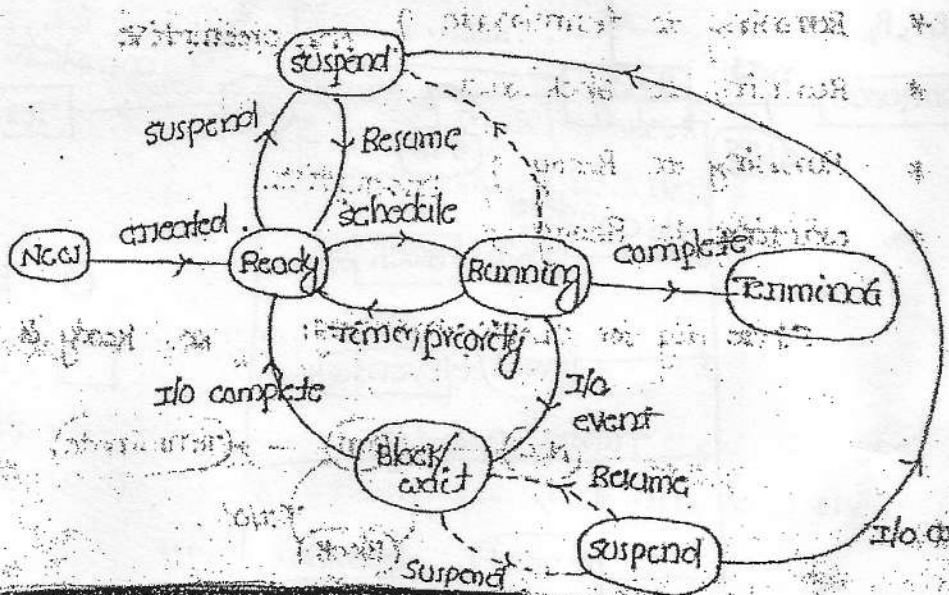
Q 13,43 Pre-emptive multiprogramming:



=

* Degree of multiprogramming = no of process.
* Degree increases, when process also increases, so we suspended process when even required.
* Process suspension is swapped out on temporary basis and later on resumed.
* There is no suspension for : → New
  → Terminate.
* The most desirable state for suspension: "Ready state"
* only Ready, Running, Block states are can be suspended.





* The state dig for multiuser, preemptive multiprogramming
* state transition dig is Deterministic finite Automata (DFA).

Eg: consider a system with n-CPU's and m-processes where n≥1 and m>n. calculate the lower bound (n≥1) and upper bound (m>n) on the process state.

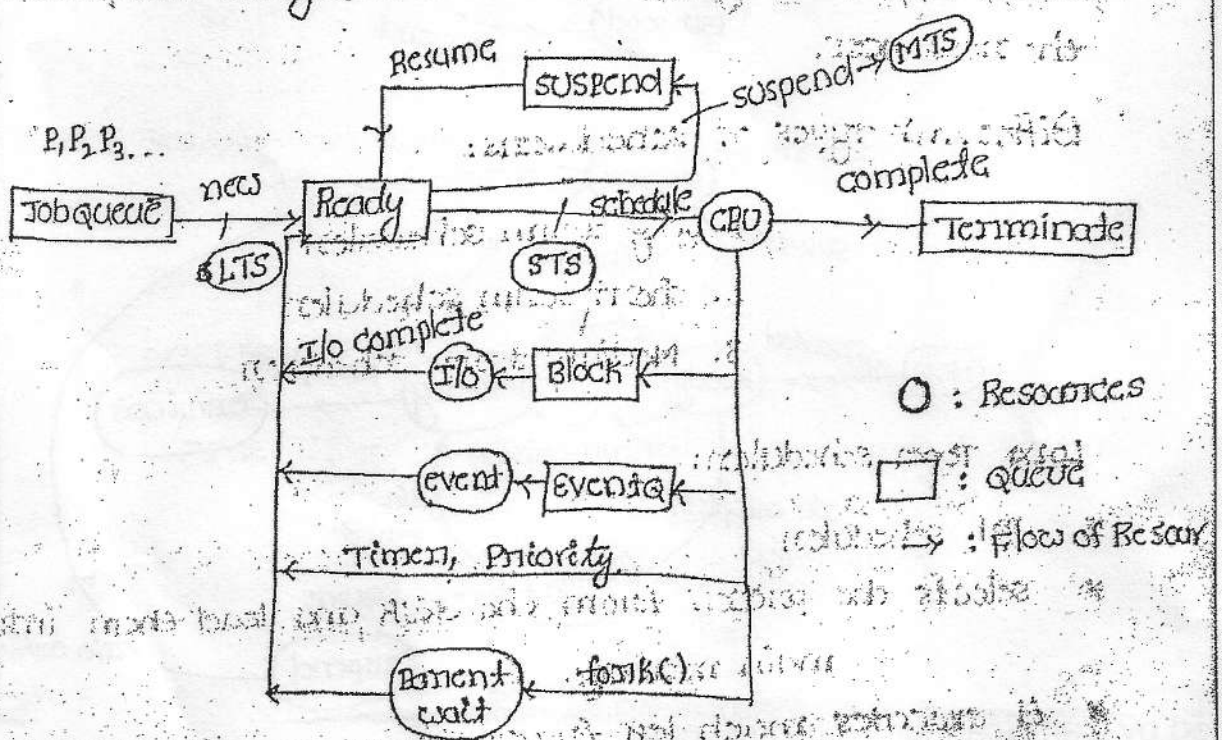
Process	Min	max
Ready	0	m
Running	0	n
Blocked	0	m.

scheduling Queues:

* it maintains information of all ready processes for CPU.
* it is maintied as a linked list.

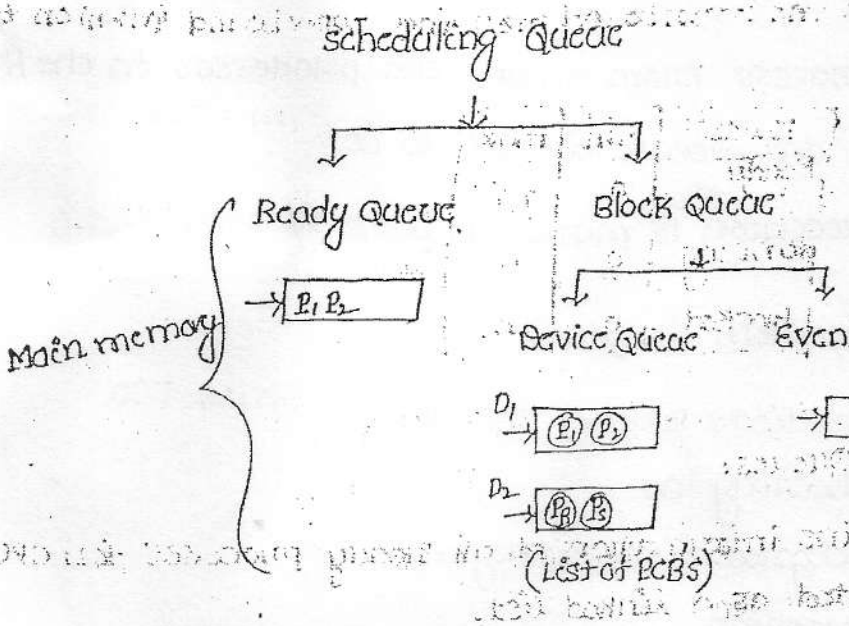
Job queue: As the processes enter the system they are put in a job queue, the queue consists of all process in the sys.

Ready Queue: The processes that are residing in main memory and are ready and waiting to execute are kept in list is called "Ready Queue".



Device Queue: It consists of process waiting for a particular I/O device.

State Queing Dig:



Scheduling Queue → Suspended Queue → Secondary Memory.
Job Queue

Scheduler: (Making Decision)

Scheduler is responsible for selecting a process for scheduling/operating with the resources, from the queues, serving the resources.

Different types of schedulers:

1. Long term scheduler
2. short term scheduler
3. Medium term scheduler

Long Term scheduler:

* Job scheduler
* selects the process from the disk and load them into main memory.
* It executes much less frequently compare to others.

* it controls the degree of multiprogramming.
* Time sharing os have no long-term scheduler.

Short Term scheduler:

* "CPU scheduler"
* it selects a process from among the processes in the RQ to execute and allocate the CPU to it.
* Frequency of execution is more compare to others.

Medium Term scheduler: (suspend)

* it is an intermediate level of scheduling
* used in timesharing os.
* Decreases the degree of multiprogramming.
* it requires swapping.

Context switch:

* switching a process to another process on CPU is called "process scheduling"
* switching the CPU from one process to another, saving the state of the old process and loading the saved state of the new process. is called as "context switching".
* CST depends on the memory speed, no.of registers,
* "Kernal uses the scheduling, dispatching and context save to control processes"

Dispatcher:

* it is a component or module of CPU scheduling function
* it gives control of the CPU to the process selected by the short term scheduler.

* Dispatcher functions
  - → switching the context
  - → switching to user mode
  - → jumping to the proper location in the user prog to restart the select prog.

* The time taken by dispatcher to stop one process and start another process is called "Dispatch latency"
  - ⇓
  - overhead => CPU overhead.

CPU scheduling Goals:

Maximize
* CPU utilization
* Throughput

Minimize
* waiting time
* Response time
* Turn around time.

Process Times:

* Arrival time / submission time (AT) (New - Ready)
* Burst time / service time (BT)
* completion time (CT)
* waiting time (WT) → I/O
  - → Ready Queue (CT - BT - AT)
* Turnaround time: (TAT) = CT - AT
* Deadline : no. of process completed within the deadline.

=

	← TAT →								
AT	BQ	BT	WT	WT	BT	WT	WT	BT	CT
	WT	CPU	I/O	BQ	CPU	I/O	BQ	CPU	

Notations:

* n - processes ($P_1 \ldots P_n$)
* $AT = A_i$

* BT = X_i
* CT = C_i
* Deadline = D_i

Formula:

* Turnaround time (TAT): $C_i - A_i$
* Avg TAT = $\frac{1}{n} \sum_{i=1}^{n} (C_i - A_i)$
* waiting time (WT): TAT - BT
  : $C_i - A_i - X_i$
* Avg WT = $\frac{1}{n} \sum_{i=1}^{n} (C_i - A_i - X_i)$
* schedule length = Max(C_i) - Min(A_i)
* Throughput = $n/t = \eta$
* Deadline overrun: $C_i - D_i$ $(C_i - D_i) < 0$ under run
  > 0 overrun
  $= 0$ on deadline.

CPU scheduling Algorithms:

Q11
* First come first serve (FCFS) } Non-preemptive
* shortest job first (SJF) }
* priority Non-preemptive }
* shortest Remaining time first (SRTF) } preemptive
* priority preemptive }
Q3 * Round Robin (Time shared OS) }
* Multilevel Queue scheduling. }
* Multilevel feedback Queue scheduling }
* Longest Remaining Time First (LRTF) => preemptive
* Highest Response Ratio (HRRN) => Non-preemptive.

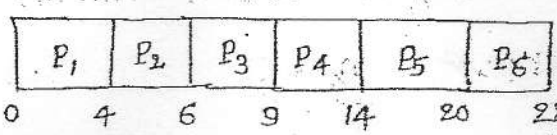
First-come-First serve:

* FCFS : Non-preemptive
* FCFS criteria is Arrival time.
* The average waiting time under the FCFS policy is often quite long.

Eg:

P.NO	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	2	6	5	3
3	2	3	9	7	4
4	3	5	14	11	6
5	4	6	20	16	10
6	5	2	22	17	15

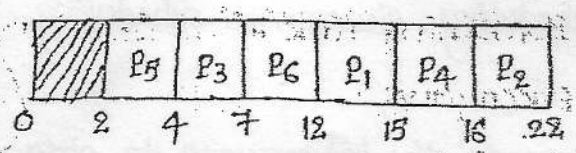
schedule length
= Max(CT) - Min
= 22 - 0
= 22



Eg:

P.NO	AT	BT	CT	TAT	WT
1	5	3	15	10	7
2	8	6	22	14	8
3	3	3	7	4	1
4	6	1	16	10	9
5	2	2	4	2	0
6	4	5	12	8	3

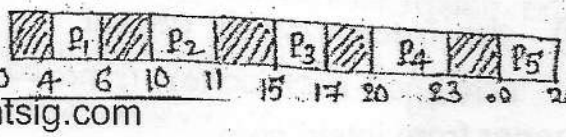
schedule length
= 22 - 2 = 20.



Eg:

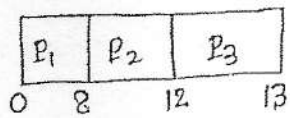
P.NO	AT	BT	CT	TAT	WT
1	4	2	6	2	0
2	10	1	11	6	0
3	15	2	17	2	0
4	20	3	23	3	0
5	28	8	36	8	0

schedule length
= 36 - 4 = 32



Q3(c)

P.NO	AT	BT	CT	TAT	WT
P_1	0.0	8	8	8	0
P_2	0.4	4	12	13.6	7.6
P_3	1.0	1	13	12	11



* when all processes arrive at same time submission time = waiting time = completion time = Turnaround time.

* FCFS may convay effect, which causes low CPU utilization.

DisAdv: starvation.

Q16

* Disk scheduling : SCAN
* Batch processing : FIFO (Queue)
* Interrupt processing : LIFO (stack)
* Time sharing : Round Robin.

Q25

* Gang scheduling : Thread scheduling
* Rate monotonic : Realtime scheduling
* Fair share scheduling : Guaranted scheduling.

Q1

* A processor needs s/w interrupts to obtain the system services which need execution of Privileged instruction.

Q2 A CPU has 2 modes - Privileged and Non-privileged. In order to change the mode from ~~non~~ privileged to Non-privileged:

A. H/w interrupt is needed.
(B) s/w interrupt is needed.
C. Privileged instruction is needed.

Shortest Job First scheduling:

Allocate the CPU to the process with the smallest or next CPU burst.

* Mode: Non-preemptive
  criteria: Burst time.
* Two processes have same next CPU burst follow FCFS.
* Also called as shortest Process Next (SPN)

Eg:

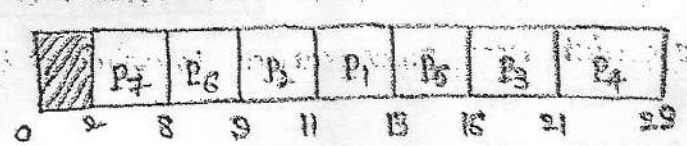
P.NO.	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	2	8	7	5
3	2	3	11	9	6
4	3	1	5	2	1
5	4	5	16	12	7
6	5	1	6	1	0

Schedule length
= 16-0 = 16.

Eg:

P.NO	AT	BT	CT	TAT	WT
1	8	2	13	5	3
2	3	2	11	8	6
3	6	5	21	15	10
4	2	8	29	27	19
5	5	3	16	11	8
6	4	1	9	5	4
7	2	6	8	6	0

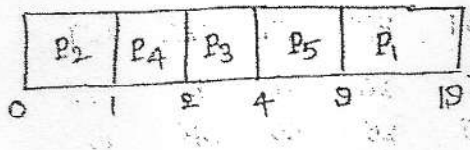
Schedule length
= 29-2 = 27



* It is the optimal CPU scheduling algorithm.
* It gives min avg waiting time.
* Frequently used in LTS, cannot be implemented in STS.
* SJF scheduling can be approximated by predicting the value of the length of the next CPU burst.

Q. * SJF scheduling algorithms will result in the maximum throughput.

P.NO	AT	BT	CT	TAT	WT
P_1	0	10	19	19	9
P_2	0	1	1	1	0
P_3	0	2	4	4	2
P_4	0	1	2	2	1
P_5	0	5	9	9	4

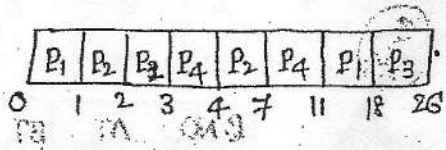


Shortest Remaining Time First (SRTF):

* MODE: Pre-emptive.
* criteria: Burst time.
* " Preemption of running process is based on the arrival of a new shorter process."

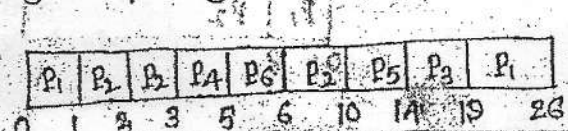
*Eg:

P.NO.	AT	BT	CT	TAT	WT
P_1	0	8	18	18	10
P_2	1	4	7	6	2
P_3	2	9	26	24	15
P_4	3	5	11	8	3



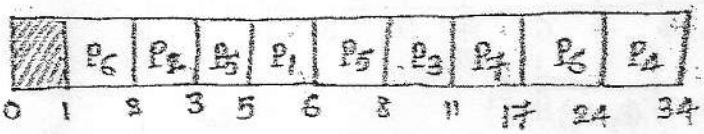
*Eg:

P.NO.	AT	BT	CT	TAT	WT
1	0	8	26	26	18
2	1	6	10	9	3
3	2	5	19	17	12
4	3	2	5	2	0
5	4	4	14	10	6
6	5	1	6	1	0



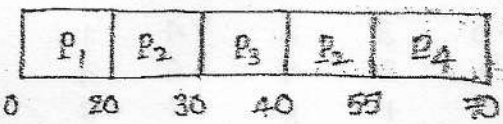
Eg:

P.NO.	AT	BT	CT	TAT	WT
1	5	1	6	1	0
2	2	1	3	1	0
3	4	3	11	7	4
4	1	10	34	33	23
5	3	4	8	5	1
6	1	2	24	23	15
7	2	6	17	15	9



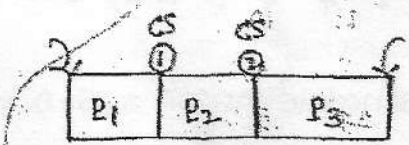
30)

P.NO	AT	BT	CT	TAT	WT.
P_1	0	20	20	20	0
P_2	15	25	55	40	15
P_3	30	10	40	10	0
P_4	45	15	70	25	10



33)

P.NO	AT	BT	CT	TAT	WT
P_1	0	10	10	10	0
P_2	2	20	30	28	8
P_3	6	30	60	54	24

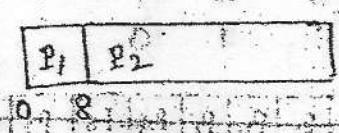


context switches = 2

[Do not count context switchs at zero and at the end]

Q 34)

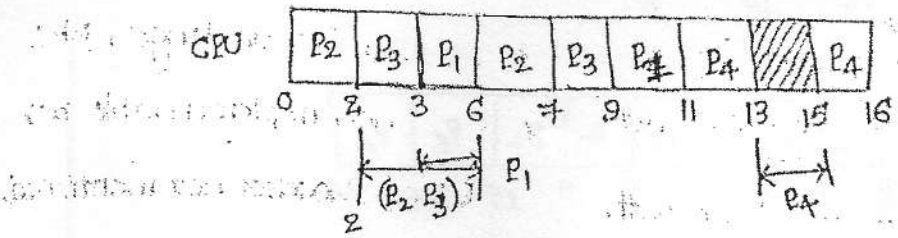
P.NO	AT	BT	CT	TAT	WT	Priority
P_1	0	2				0
P_2	0	4				1
P_3	0	8				2



Eg:

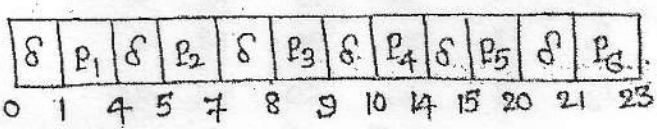
P.NO	AT	BT	IOBT	BT
1	0	(3	2	2)
2	0	(2	4	1)
3	2	(1	3	2)
4	5	(2	2	1)

RQ: P2/1 P3/2 P4/2 P1/2



Eg: FCFS CPU scheduling overhead $\delta = 1$ unit

P.NO	AT	BT	CT	TAT	WT
1	0	3	4	4	1
2	1	2	7	6	4
3	2	1	9	7	6
4	3	4	14	11	7
5	4	5	20	16	11
6	5	2	23	18	16



schedule length = 23 − 0 = 23.

% CPU time overhead = $6/23 \times 100$

% CPU time useful = $1 - 6/23 = 17/23 \times 100$

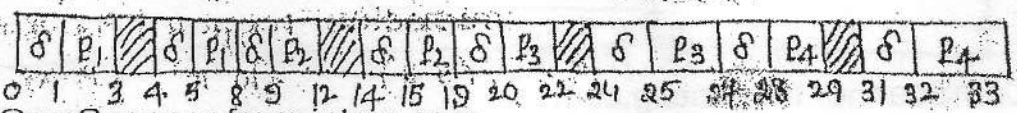
FCFS:

Eg $\delta = 1$, IOBT/IOWT is non zero.

P.NO.	AT	BT	IOBT	BT
1	0	(2	1	3)
2	1	(3	2	4)
3	2	(2	2	2)
4	3	(1	2	1)

% CPU idleness =

% CPU useful =



Performance of Shortest Job first: "Favours of Shorter jobs"

Burst time

Advantages

(OPTIMAL ALGO)
* Maximum Throughput
* Minimum Avg W.T & TAT
* It is implementable with Predicted/estimated BT

Disadvantages

* Starvation to longer jobs.
* It is non implementable bcz BT of Process not mentioned.

=

SJF with predicated Burst Time:

Prediction Techniques

Static
1. Process Size
2. Process Type

Dynamic
1. Simple averaging
2. Exponential average/'Aging'

Process Size (Bytes):

Based on past records burst time, next burst time can be predicted for new records, functional change may present.

Let P = 201 KB

∴ P_{old} = 200 KB ⇒ 20 units of time

∴ P_{new} = 201 KB ⇒ 20 units of time.

Process Type:

Process

High priority ⇐ OS (3-5 units)

User

Interactive (5-8) | Foreground (10-15) | Background (15-20) | Batch (20-40)

Dynamic prediction Tech:

1. Simple Average (or):

   → Given n-processes ($P_1 \ldots P_n$).

   → Let 't_i' be the actual BT.

   → Let "τ_i" denotes the predicted B.T.

NOTE: "Next CPU BT of the processes is the average of Process completed Burst times."

$$\tau_{n+1} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

	BQ	CPU	IO	BQ	CPU	BQ	CPU
AT	ωT_1	BT	ωT_2	ωT_3	BT	ωT_4	?
	τ_1	t_1	τ_2	τ_3	t_2	τ_4	τ_{n+1} t_3

τ_1 = initial gues/seed values.

2. Exponential Average / Aging Algorithm:

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

$0 \le \alpha \le 1$

* if $\alpha = 0 \Rightarrow \tau_{n+1} = \tau_n$
* if $\alpha = 1 \Rightarrow \tau_{n+1} = t_n$
* if $\alpha = 0.5 \Rightarrow \tau_{n+1} = \frac{1}{2}(t_n + \tau_n)$

$$\Rightarrow \tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \rightarrow (1)$$

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

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

$$= \alpha t_n + \alpha(1-\alpha)t_{n-1} + \alpha(1-\alpha)^2 t_{n-2} + (1-\alpha)^3 \tau_{n-1} \rightarrow (3)$$

Eg: $\alpha = 0.5$, $\tau_1 = 10$ process: 4 8 6 7 then τ_5
$t_1\ t_2\ t_3\ t_4$

$\Rightarrow \tau_5 = \frac{1}{2}(t_4 + \tau_4) \Rightarrow \tau_4 = \frac{1}{2}(t_3 + \tau_3) \Rightarrow \tau_3 = \frac{1}{2}(t_2 + \tau_2) \Rightarrow \tau_2 = \frac{1}{2}(t_1 + \tau_1)$

$= \frac{1}{2}(7 + 6.75)$ $= \frac{1}{2}(6 + 7.5)$ $= \frac{1}{2}(8 + 7)$ $= \frac{1}{2}(4 + 10)$

$\tau_5 = 6.875$ $\tau_4 = 6.75$ $\tau_3 = 7.5$ $\tau_2 = 7$

=

Sol

$\alpha = \frac{1}{2}$ $\tau_1 = 10$, Process: 40 20 40 15 then next prediction
$t_1\ t_2\ t_3\ t_4$

$\Rightarrow \tau_5 = \frac{1}{2}(t_4 + \tau_4) \Rightarrow \tau_4 = \frac{1}{2}(t_3 + \tau_3) \Rightarrow \tau_3 = \frac{1}{2}(t_2 + \tau_2) \Rightarrow \tau_2 = \frac{1}{2}(t_1 + \tau_1)$

$= \frac{1}{2}(15 + 31.25)$ $= \frac{1}{2}(40 + 22.5)$ $= \frac{1}{2}(20 + 25)$ $= \frac{1}{2}(40 + 10)$

$= 23.125$ $\tau_4 = 31.25$ $\tau_3 = 22.5$ $\tau_2 = 25$

$$\tau_{n+1} = \frac{1}{n}\sum_{i=1}^{n} t_i = \frac{1}{4}(40 + 20 + 40 + 15) = \frac{115}{4}$$

$$= 28.75$$

=

Longest Remaining Time First (LRTF):

Mode: ~~Burst time~~ Preemptive

criteria: Burst time

NOTE: "Ties favouring process having lower process id."

Eg:

P.NO.	AT	BT	CT	TAT	WT
0	0	1	12	12	10
1	0	1	13	13	9
2	0	1	14	14	6

CPU: | P_2 | P_1 | P_2 | P_1 | P_2 | P_0 | P_1 | P_2 | P_0 | P_1 | P_2 |

0 4 5 6 7 8 9 10 11 12 13 14

Avg TAT = $\frac{12+13+14}{3} = 13.00$

Highest Response Ratio (HRRN):

Criteria: Response Ratio (R.R) = $\frac{w+s}{s}$

w: waiting time for a process so far.

s: service time of a process. on BT

Mode: Non-preemptive

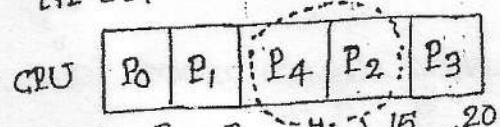
NOTE: "HRRN not only favours shorter jobs but also limits the waiting time of longer jobs."

* schedules the job having highest response ratio and run till its completion.

Eg:

P.NO	AT	BT	CT	TAT	WT
0	0	3	3	3	0
1	2	6	9	7	1
2	4	4	13	9	5
3	6	5	20	14	9
4	8	2	15	7	5

NP SJF



HRRN

Difference

RP_2: $\frac{w+4}{4}$ $w = (9-4) = 5$

$= 5+4/4 = 2.25$

RP_3: $\frac{3+5}{5} = \frac{8}{5} = 1.6$

RP_4: $\frac{1+2}{2} = \frac{3}{2} = 1.5$

$\therefore P_2$ has HRR to choose

Next, we need to choose b/w P_4 and P_3, Hence need to calculate Response Ratio again.

RP_3: $\frac{(13-6)+5}{5} = \frac{12}{5} = 2.4$

RP_4: $\frac{(13-8)+2}{2} = \frac{7}{2} = 3.5$

RP_4 has HRR to choose P_4

Priority based scheduling:

Priority
- Static — Not changed throughout its execution.
- Dynamic — changed at regular interval of time

Note: Increasing the priority level regularly at the run-time is called as "Aging Algorithm."

Priority: Starvation.

Non-preemptive priority scheduling:

Criteria: Priority

Mode: Non-preemptive

Eg:

P.No	Priority	AT	BT	CT	TAT	WT	HT (Hold time)
1 (i)	2	0	4	4	4	0	0
2	4	1	2	25	24	22	22
3	6	2	3	23	21	18	18
4	10	3	5	9	6	1	1
5	8	4	1	20	16	15	15
6	12 (ii)	5	4	13	8	4	4
7	9	6	6	19	13	7	7

Preemptive priority scheduling:

Criteria: Priority

Mode: Preemptive.

Eg:

P.No	Priority	AT	BT	CT	TAT	WT
1	2	0	4	25	25	21
2	4	1	2	22	21	19
3	6	2	3	21	19	16
4	10	3	5	12	9	4
5	8	4	1	19	15	14
6	12	5	4	9	4	0
7	9	6	6	18	12	6

Round Robin scheduling:

Criteria: AT+ Time Quantum

Mode: Pre-emptive.

* Multiprogram / Multiuser / Time shared operating system.
* Similar to FCFS but preemption is added to switch b/w process
* Ready queue => circular queue.

Eg: TQ = 2

P.NO	AT	BT	CT	TAT	WT
1	0	4	8	8	4
2	1	5	19	18	13
3	2	6	23	21	15
4	3	3	18	15	12
5	4	1	11	7	6
6	5	4	21	16	12

RQ: $P_1 P_2 P_3 P_1 P_4 P_5 P_2 P_6 P_3 P_4 P_2 P_6 P_3$

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

0 2 4 6 8 10 11 13 15 17 18 19 21 23

Performance of RR:

* TQ is very small, efficiency = 0 bcz most of the time is wasted in context switching.
* TQ is small => context switching is more => overhead high. ie improves Responsiveness.
* TQ is large => context switching is ~~high~~ less => overhead less ie Degenerates Responsiveness.

Eg: TQ = 3

P.NO	AT	BT	CT	TAT	WT
1	~~0~~ 7	3	21	14	11
2	4	7	35	31	24
3	6	5	32	26	21
4	2	4	15	13	9
5	5	6	30	25	19
6	3	8	34	31	23

RQ: $P_4 P_6 P_2 P_5 P_4 P_3 P_1 P_6 P_2 P_5 P_3 P_6 P_2$.

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

0 2 5 8 11 14 15 18 21 24 27 30 32 34 35

Q. consider 'n' processes sharing the CPU in RR fashion. Assuming that each process switch takes s seconds, what must be the quantum size 'q' such that the overhead resulting from process switching is minimized, but at the ~~at the~~ same time each process is guaranteed to get its turn at the CPU at least every 't' seconds.

A. $q \leq \frac{(t-ns)}{(n-1)}$ B. $q \geq \frac{(t-ns)}{(n-1)}$ C. $q \leq \frac{(t-ns)}{(n+1)}$ D. $q \geq \frac{(t-ns)}{(n+1)}$

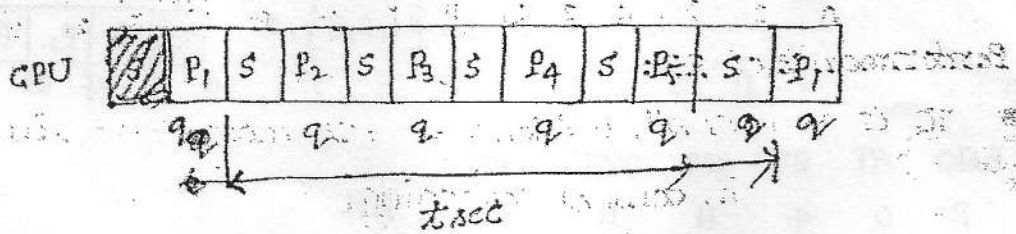
→ n processes ($P_1 \ldots P_n$)

→ context switch time = s sec

∴ n process context switch time = ns sec.

Time quantum = q

n = 5



case (i) ⇒ $TQ * (n-1) + sn \geq t$

At least after t sec ⇒ $q * (n-1) \geq (t - sn)$

$$q \geq \frac{(t-sn)}{(n-1)}$$

Here P_1 gets its turn at CPU after 't' sec.

In between $t = (n-1)$ time quantum par along with switching.

case (ii): Exactly after 't' sec ⇒ $q = \frac{(t-ns)}{(n-1)}$

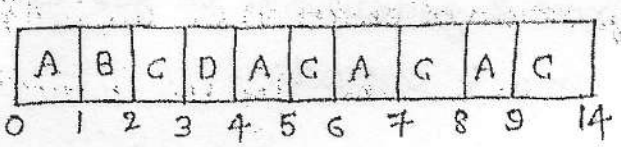
case (iii) Atmost after 't' sec ⇒ $q \leq \frac{(t-ns)}{(n-1)}$

Q3

P.NO	AT	BT	CT	TAT	WT
A	0	43	9	9	5
B	0	10	2	2	1
C	0	87	14	14	6
D	0	10	4	4	3

RR
TQ = 1 unit

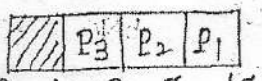
completion Time of A = 9



Q4 The sequence — is an optimal SJF scheduling sequence for the following jobs which leaves the CPU idle for — units of time

P.NO	AT	BT
1	0.0	9
2	0.6	5
3	1.0	1

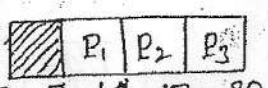
(A) {3, 2, 1}, 11



B. {2, 1, 3}, 0 : NOT POSSIBLE

C. {3, 2, 1}, 0 : NOT POSSIBLE

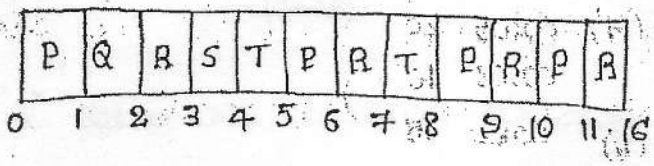
D. {1, 2, 3}, 5



Q5

P.NO	AT	BT	CT	TAT	WT
P	0	4	11	11	7
Q	0	1	2	2	1
R	0	8	16	16	8
S	0	3	4	4	3
T	0	2	8	8	6

RR
TQ = 1



Q6 which of the following does not Interrupt a Running process.

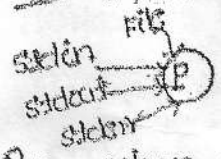
(A) A device : I/O interrupt

(B) A Timer : Timer interrupt

✓(C) scheduler process : Never interrupt

(D) Power failure : one type of interrupt

Q 29. I/O redirection:



can be employed to use an existing file as input file for a program

Q 30. when an Interrupt occurs, an OS:

Always resumes execution of interrupted process after processing the interrupt.

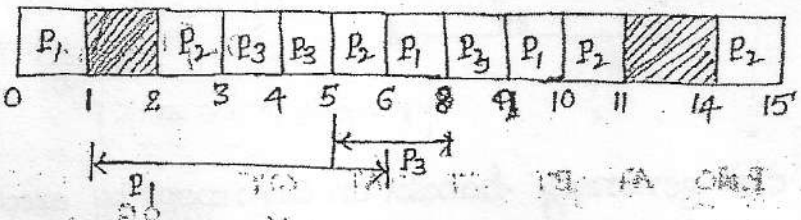
Q 31. An example of spooled Device

A linear printer used to print the o/p of no. of jobs.

Q 32.

P.NO	AT	Priority	(CPU	I/O	CPU)
P_1	0	2	1	5	3
P_2	2	3 (L)	3	3	1
P_3	3	1 (H)	2	3	1

Preemptive priority

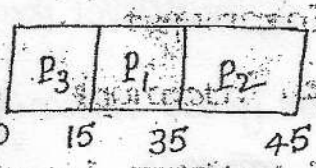


Complete times of the processes: P_1 P_2 P_3 are 10, 15, 9 respectively.

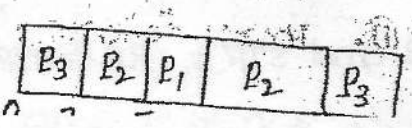
Q 33.

P.NO	Priority	AT	BT
P_1	10 (H)	00:05	20
P_2	9	00:03	10
P_3	8 (L)	00:00	15

Non-Preemptive priority:



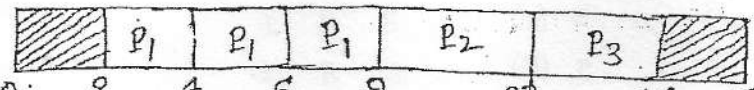
Ans Preemptive priority:



Q3c Three processes arriving at time zero. with total execution time of 10, 20 and 30 units respectively. Each process spends the first 20% of execution time doing I/O, 70% time doing computation & last 10% of time doing again I/O.

⇒ SRTF % CPU remain idle.

PNO	AT	(IO	BT	IO)
P_1	0	(2	7	1)
P_2	0	(4	14	2)
P_3	0	(6	21	3)



0 2 4 6 9 23 44 47

P_2 P_1 P_2

P_3

% CPU remain idle $= \frac{5}{47} \times 100 = 10.6\%$

Q4c A CPU generally handles an interrupt by executing an interrupt service routine: By checking the interrupt register after finishing the execution of current instruction.

Q: what is the relation holds b/w the following pairs of set of algorithm.

* Priority and SJF : SJF gives higher priority to lower BT, Hence it is actually a priority queue.

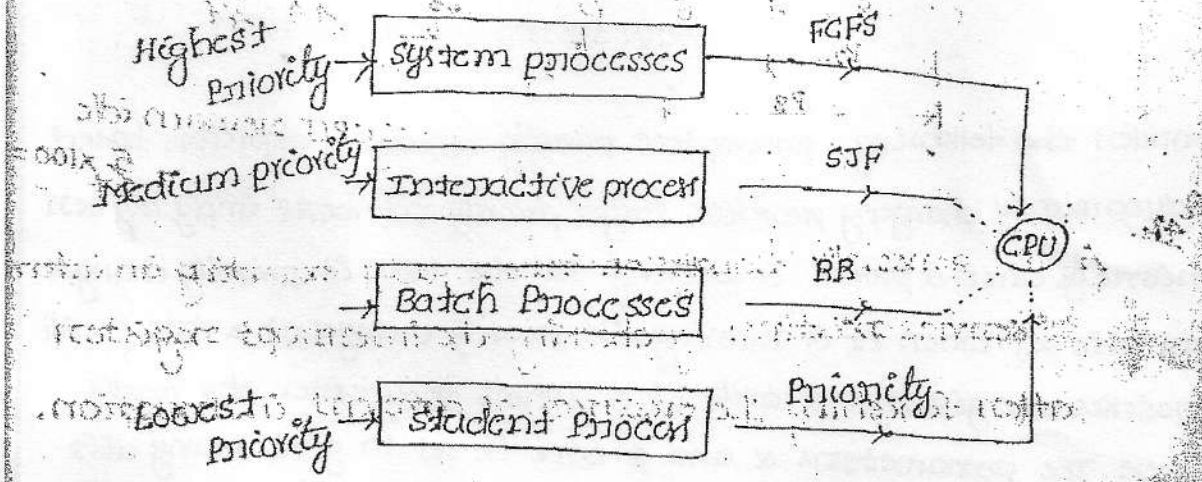
* Multilevel Feedback Queue and FCFS: To implement FCFS, we need to maintain a multilevel feedback mech.

* Priority and FCFS: FCFS gives priority to arrival time of each process.

* RR and SJF : RR ≠ SJF ⇒ no relation exists.

Multilevel Queue scheduling:

* Multilevel queues are an extension of priority scheduling where by all processes of the same priority are placed in a single queue.
* Time sharing systems often supports an idea of foreground and background processes.
* Foreground process service an interactive user.
* Background process are intended to run whenever no foreground proces requires the CPU.



Dis Adv: The second queue is service only when first queue is empty/complete

* Starvation.

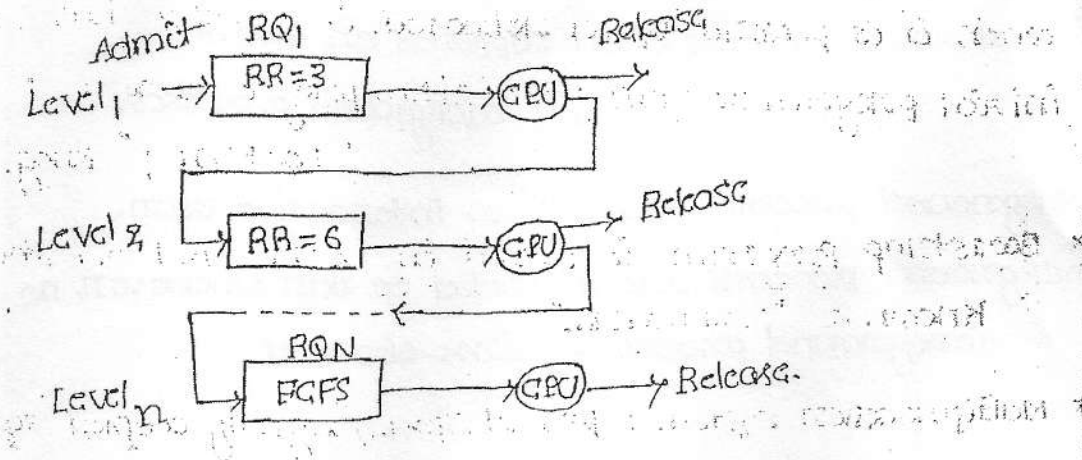
Multilevel Feedback Queue scheduling:

* This overcomes the disadvantages of multilevel queue scheduling.

Parameters:

* The no. of queues.
* scheduling algorithm for each queue.
* Method used to determine when to demote a process to a lower priority queue.

* upgrade a process to higher priority queue.
* which queue a process will enter when that process needs services.



Adv: overcome starvation.

consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU, its priority changes at a rate α; when it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many diff scheduling algor.

A. what is the algorithm that results from $\beta > \alpha > 0$

α : priority rate for process waiting in RQ.

β : Priority rate level of running process.

$\beta > \alpha > 0 \Rightarrow$ Running process priority is always greater than priority of waiting processes in RQ $\Rightarrow$ FCFS.

B. what is the algorithm that results from $\alpha < \beta < 0$.

$\alpha < \beta < 0 \Rightarrow$ Process waiting in RQ will always be more than process running. Hence, the new processes will have more priority than the running processes $\Rightarrow$ LIFO.

NOTE:* systemcall provides an interface to the service made available by OS.

* Bootstrap: For a computer to start running for instance when it is powered up or rebooted, it needs to have an initial program to run. This initial prog is known as "Bootstrap" Prog.

* Bootstrap program is stored in ROM or EEPROM which is known as "Firmware."

* Multiprocessor system: Parallel system/tightly coopled sys.

* cluster system: computers share storage and closely linked via a LAN

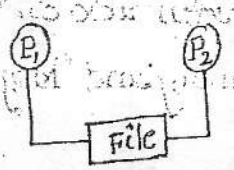
* SPOOL: simultaneous peripheral operation-on-Line.

PROCESS MANAGEMENT - II

- Interprocess communication
- synchronization
- concurrency & control
- Deadlock.

Interprocess communication: For communication among any two process, that may be two computers, (or) two cell phones (or) any two electronic devices (or) any two entities in the world (or) universe, there must be a media to communicate.

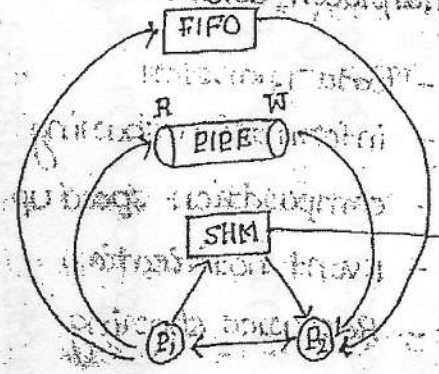
Eg: File (File is simple media not used always)



The communication can be done b/w P_1 and P_2 by using either s/w and H/w media

H/w media: wired, cable, wireless, NIC.

s/w media: Protocols.



simple message passing sys like email

* Equivalance of file in main memory with Read and write operations, is Pipe.

* The popular interprocess communication schemes are
  - → shared memory
  - → Msg Memory

Processes

Intra process Inter process

Intra processes: with in a single process communication.

How to communicate a single process

Eg:
```
main()
{
}
fun()
{
}
```

The single process can communicate by using → parameter passing
→ Global variable.

Inter processes: communication between two or more proces by using shared memory and Msg memory.

IE: No shared resources ⇒ No interprocess communication
⇓
There is no synchronization.

Eg: No program there is no meaning of proc

Purpose for interprocess communication:

- Data transfer
- information sharing
- computation- speed up
- Event notification
- Resource sharing
- process control
- synchronization.
- Modularity
- convenience.

∴ No interprocess communication ⇒ No synch

What is synchronization?

Need for synchronization?

How do achieve synchronization?

Process synchronization:

* An important and fundamental feature in modern OS is concurrent execution of processes.
* Process synchronization is essential for the realization of
  - multiprogramming
  - multiprocessing
  - distributed system
  - client-server model of computation
* In the multiprogramming sys, some processes performs read and some processes perform write on the file simultaneously.
* This lead to data inconsistency as data is being read as well as modified by many process at the same time.
* To prevent such data inconsistency process synchronizat is required.
* All process that want to perform read operation and can do the reading simultaneously but process that needs to perform write operation should do it one at a time.
* The processes that exist at the same time are called "concurrent process".

Problems due to lack of synchronization or) Need for synch:

- Inconsistency => incorrectness (wrong result)
- Loss of data
- Deadlock.

How to achieve synchronization:

Process synchronization

competing sync proces co-operating sync proces

competing sync proces: Two or more processes are said to be in competing synchronization iff they complete on content for the accessability of shared resources.

Eg: Two processes A and B try to update the value of a shared variable.

* In general lack of competion synchronization among processes leads to the problems of → inconsistency
→ loss of data.

Co-operating synchronization proccess:

Two or more processes are said be in co-operating synchronization iff execution of one proces affect the other processes. i.e Processes get affected by each other.

Eg: Producer-consumer problem

* In general lack of co-operating synch leads to the Problem of deadlock.

NOTE: * Producer-consumer : inconsistency Problem
* printer spooler daemon : loss of data

The producer-consumer problem (cooperating process):

* The producer produces some items and keep them into the buffer.
* consumer consumes the items produced by the Producer from the buffer.
* If the buffer is FULL, then producer must wait for atleast one location of the buffer to be empty, That is consumer's behaviours affects producer.

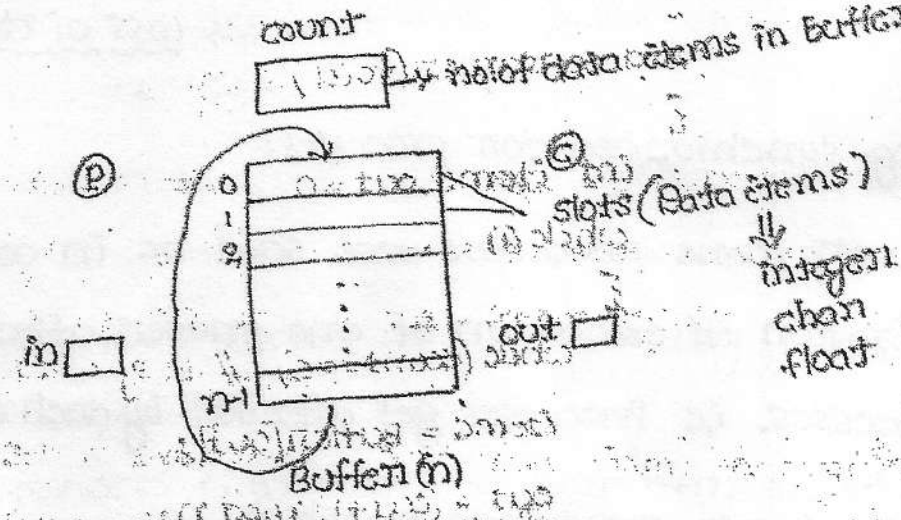
⇓

Producer get affected.

* If the buffer is EMPTY, then the consumer must wait for at least one location of the buffer to be filled up again, Thus producer's behaviours affects consumer.

⇓

consumer get affected.



* Buffer is a circular list (Bounded Buffer)
* Producer uses a variable 'in' to indicate on which location it is putting a new item.
* consumer uses a variable 'out' to indicate on which location it is going to consume an item.
* "counter" indicates no of elements.

Buffer => 0 => EMPTY, Buffer => n => FULL.

Implementation:

```
#define n 100

int Buffer[n]

void producer(void)
{
    int itemp, in=0, count=0;
    while(1)
    {
        Produce item(itemp);
        while(count==n); // Busy wait
        Buffer[in] = itemp;
        in = (in+1) mod n;
        count = count++;
    }
}

void consumer(void)
{
    int itemc, out=0
    while(1)
    {
        while(count==0); // Busy wait.
        itemc = Buffer[out];
        out = (out+1) mod n;
        count = count-1;
        Process item(itemc);
    }
}
```

Note: * Producer-consumer code suffers from Inconsistent
* Lack of synchronization in producer-consumer

IN PRODUCER

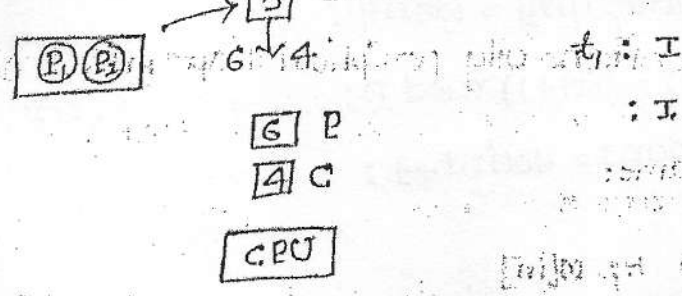
count = count+1; // To fetch the variable "count" & then perform the operation

1. LOAD R_p m[count];
2. INC R_p;
3. Store M[count], R_p;

IN CONSUMER:

count = count-1;

1. LOAD R_c, M[count];
2. DEC R_c;
3. STORE M[count], R_c;



In producer: while first two steps get executed, preemption occurs, i.e consider the count = 5, In step, the value is loaded into reg R_p and step 2 the value in the register get incremented so the value of count = 6.

Before storing the "count" value from the register it go to preempted so the producer have count value = 6.

In consumer: count value = 5, stored in a register R_c and then decremented, before transferring the value from "R_c" it preempted. so it have count = 4.

∴ There is inconsistency bcz count value is never equal to 5.

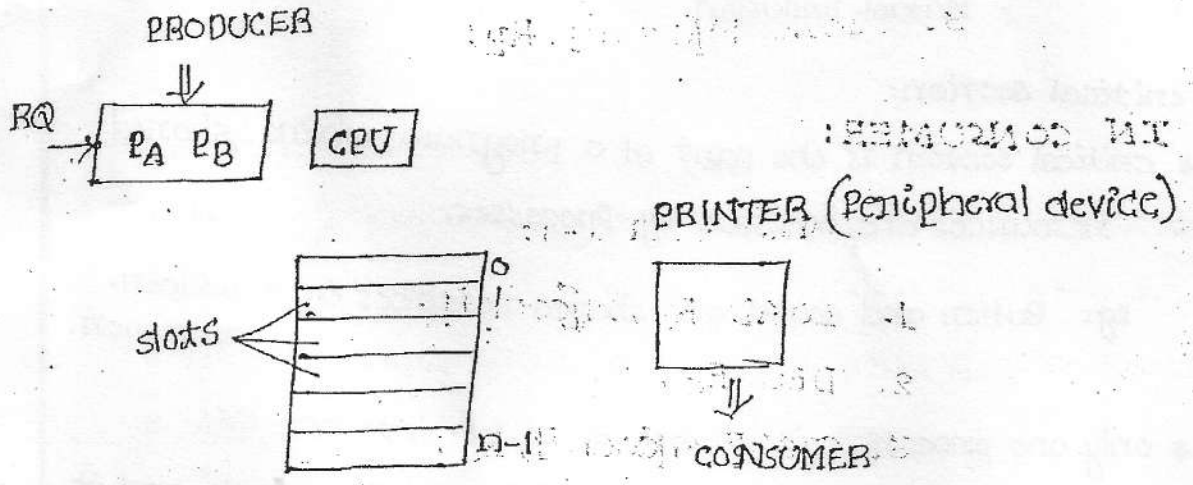
PRINTER SPOOLER DAEMON:

⇓

Background process

⇓

Running infinite proces => Providing service

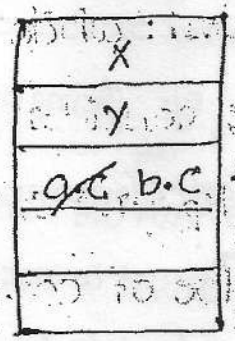


spooler directive

SPOOL: Simultaneously peripheral operation on line.

Insert-filename:

1. LD R_i M[in]
2. store SD[Ri], "FN"
3. INC R_i
4. store M[in], R_i



T_i : P_A : I, II, III S.D

→ Preemption

T_j : P_B : I, II, III, IV

Problem: Loss of data.

Necessary conditions for synchronization:

- critical section & non-critical section
- Race condition (concurrency)
- Pre-emption
- Mutual Exclusion.

critical section:

* critical section is the part of a program where shared resources are accessed by processes.

Eg: Buffer and count are shared resources in producer-consumer.

* only one processes is allowed to enter into the CS.
* critical section is also called as "storage interlock" and it is a H/w features of a shared memory.
* Critical section => shared Resource (any variable/device)
* when many processes ~~allowed to enter~~ try to accesses the same data item, then this feature sequentializes all such requests and executes one by one.
* Non critical section is part of program which do not access shared resources.

Eg: itemp and itemc in producer-consumer.

Race condition: (concurrency)

* Processes must be racing together to access a shared resources/critical section.
* Race condition implies concurrent execution.
* Race condition situation occurs several processes access and manipulate shared data concurrently.
* The order of execution of instruction influences the result produced.

Pre-emption:

A process get executed in the middle of other process

count = count+1

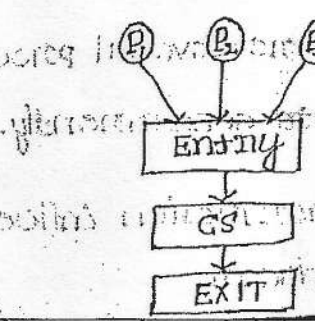
① LOAD
② INC
③ STORE.

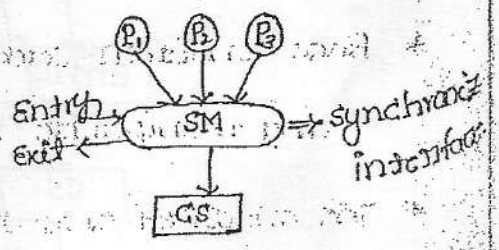
Mutual Exclusion:

* No two processes may be present in the critical section at the same time to access the shared resources.
* No assumption regarding to the relative speed of Processes.
* A process in its cs for a finite time only.
* processes requesting access to cs should not wait indefinitely
* A process waiting to enter cs can not be blocking a process in cs.

Synchronization Mechanism:

* Mutual Exclusion ensures that there is no problem of Race condition among processes.
* process should not be alloced to access critical section directly.
* proces must be request permission to enter the cs.
* Entry section must ensure the MUTUAL EXCLUSION Principle
* Process must notify when leaving the region.
* Exit section informs the available of the critical section to other waiting processes.





* There are total 8 synchronization mechanisam.
* 8 synchronization mechanisams are classified into 2 types.

1. Mutual Exclusion with busy waiting (or) spin lock
2. Mutual Exclusion without busy waiting (or) locking.

* The two typer are implementable by using s/w on H/w on os
  ⇓ ⇓
  UM KM.

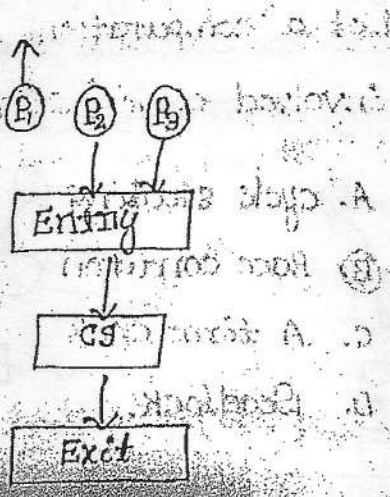
Requirements of synchronization Mechanism:

1. Primary conditions (Mandatory)
   - Mutual Exclusion (ME)
   - progress
2. secondary condition (optional)
   - Bounded waiting
   - portability (or) Anchitectural Neutral

Primary conditions

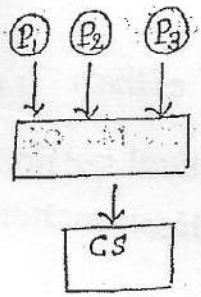
Mutual Exclusion: Mutual Exclusion must always be guaranted irrespective of weather preemption takes place or not.

Progress: No process running outside cs should block. when cs is free (other interested process from accessing/entering the cs)



secondary conditions:

Bounded waiting: No process has to wait forever to access its critical section.



If P_1 enter into the cs and after leaving P_2 wants to enter into cs and performing operations, P_2 leaves, then again P_1 wants to enter along with P_3, where P_1 has high priority than P_3 then P_1 enters into cs and then P_2 and soon where there cs no chance for P_3 to enter into cs.

DisAdv: starvation.

Portability: No assumptions about speed and no.of CPU's may be considered.

⇓

implement at UM

⇓

independent of CPU or independent of Processes

⇓

Architectural neutral

when the result of a computation depends on the speed of the proceses involved there is said to be.

A. cycle stealing

(B) Race condition

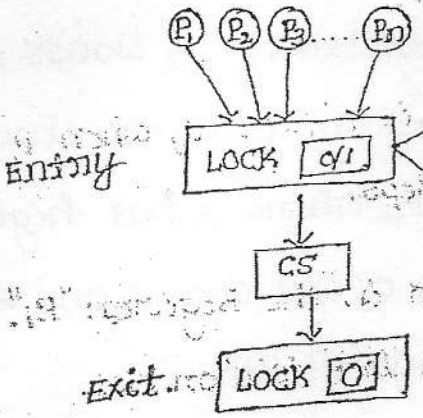
C. A time clock

D. Deadlock

1. LOCK VARIABLE:

* Software mechanism implemented in user mode.
* Mutual Exclusion with busy waiting.
* Multiprocesses (More than two process)(≥ 2)

* Lock is a shared binary variable either 0 or 1
  - Lock = 0 => critical section is free
  - = 1 => critical section is in use.



* processes has to check the LOCK value before entering into CS.
* if LOCK value = 1 => Busy waiting. if LOCK = 0, processes enter into CS and make LOCK = 1

High Level Language Implementation:

```
int LOCK = 0;
void Entry section(int process)
{
    while (LOCK != 0) ; // Busy waiting
    LOCK = 1;
    CS
}
void exitsection (int process)
{
    LOCK = 0
```

Assembly Language:/Machine Language:

Entry section

1. LOAD R_i m[LOCK]
2. CMP R_i #0
3. JN≠ $Step_1$.
4. STORE m[LOCK], #1
5. CS
   EXIT
6. STORE m[LOCK], #0.

Here, initially the lock value is '0' (lock=0) when P_1 needs to enter CS, it follows all the steps.

$Step_1$: The lock value is stored in the Register "R_i".

$Step_2$: It is compared whether lock is '0' or not.

$Step_3$: If lock value is not zero (if lock=1) then again $Step_1$

$Step_4$: If lock=0, then the process enter CS and while enter then it changes lock = 1.

RQ: P_1 P_2 P_3

Ø 1 Lock

Entry section

CPU Ø1 | CS P_1 P_2

P_1: 1 2 3 Preempt

P_2: 1, 2, 3, 4, 5

P_3: 4, 5

- it fails to guarantee Mutual Exclusion.
- Busy waiting results in wastage of CPU cycles

Hint for testing supports to guarantee Mutual Exclusion:

* Identify the shared variable (lock)
* Read the value of shared variable
* compare (optional)
* update (based on result of comparision).

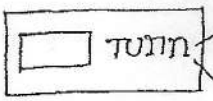
⇒ If Process P_1 entered into critical section and before changing lock=1, it get pre-empted. since the lock=0 another process P_2 wants to enter cs and get resides in CPU and get into cs.

since P_1 and P_2 gets into critical section simultaneously it fails to guarantee mutual exclusion.

⇒

2. STRICT ALTERNATION APPROACH:

* software mechanism implemented at usermode.
* Mutual Exclusion with busy waiting.
* 2-processes solution: (P_0, P_1) or (P_i, P_j)

* 

* TURN => shared variable, it has two value i.e 0 and 1
  i j
* value of turn is checked by processes P_0 and P_1.
* Turn = 0 => only P_0 can enter into CS, while leaving the CS, P_0 sets turn to 1.
* Turn = 1 => only P_1 can enter into CS, while leaving the CS, P_1 sets turn to 0.
* strictly on alternate basis, processes (P_0 and P_1) takes turn to enter CS.

Algorithm:

For P_0

```
int turn;
void P0(void)
{
   while(1)
   {
   1. Non-CS()
Entry : 2. while (turn != 0);// Busy waiting.
   3. CS
EXIT : 4. turn = 1;
   }
}
```

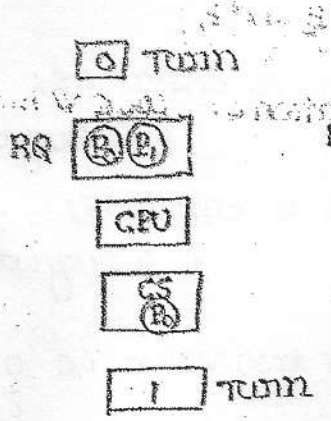
For P_1

```
void P1(void)
{
   while(1)
   {
   1. Non-CS()
Entry : 2. while (turn != 1)
   3. CS
EXIT : 4. turn = 0;
   }
}
```

turn value is not immediately update, but it is updated at EXIT section

∴ It guarantees Mutual Exclusion.

→ ME



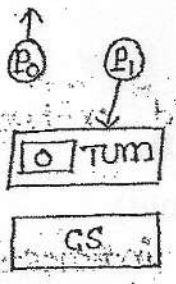
⇒ Guarantees Mutual Exclusion.

⇒ Not guarantee progress as it does not allow a process to enter critical section any time and any num of times as long as critical section is free.

Direct progress:

initially, turn = 0, P_0 is not interested enter into CS, P_0 is running in non-CS, but P_1 is interested to enter into CS but turn = 0

Hence progress is not guarnted always.



Indirect progress:

P_0 enters into the CS and while exiting turn is changed to 1, P_1 needs to enter, if again 'P_0' wants to enter CS and P_1 is not interested. P_0 can not enter into CS bcz turn = 1. Hence progress is not guarantee always.

3. PETERSON'S SOLUTION (Dekker's Algor)

* software mechanism implemented at user mode.
* Mutual Exclusion with Busy waiting
* 2- process solutions (P_0 and P_1)

* peterson's solution is combination of LOCK VARIABLE + STRICT.

Algorithm:

```
#define N 2
#define TRUE 1
#define FALSE 0

int interested[N] = FALSE;
int turn;

void Entry-section(int process)
{
  1. int other;
  2. other = 1- process;
  3. interested[process] = TRUE
  4. turn = process;
  5. while(interested[other] == TRUE && turn == process); Busy wait
}

void exitsection(int process)
{
    interested[process] = FALSE;
}
```

NOTE:

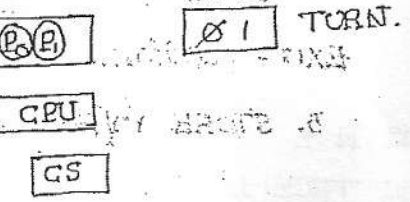
* Mutual Exclusion is guarantee.
* Progress is guarantee.
* Bounded waiting is guaranteed (it is not guaranteed for

DisAdv:

* solution available only for 2 processes.
* Busy waiting : wastage of CPU cycles.

=

```
int id[0] = F T
int id[1] = F T
```



t_i : P_0 : 1, 2

t_j : P_1 : 1, 2

t_k : P_0 : 3, 4

t_l : P_1 :

4. TEST AND SET LOCK :

* software mechanism implemented at user mode.
* Mutual Exclusion with busy waiting.
* Multi-processes solution.

TSL Register Flag → 0 ⇒ critical section is free
→ 1 ⇒ critical section is in use.

* TSL is a privileged instruction.
* one atomic operation at a time.
* Execution will be done without preempting.
* TSL has the advantages over peterson's solution, as it is applicable for more than 2 processes.

Semantics: - copy the value of flag into register
- set the value of Flag to 1
(or)

* Test and set (a,b)

1. Read b
2. copy b into a
3. set b = TRUE

Algorithm:

Entry - section

1. TSL Reg$_i$, m[Flag];
2. CMP Reg$_i$, #0;
3. JNZ step$_1$
4. CS

Exit - section.

5. STORE m[Flag], #0;

Drawback:

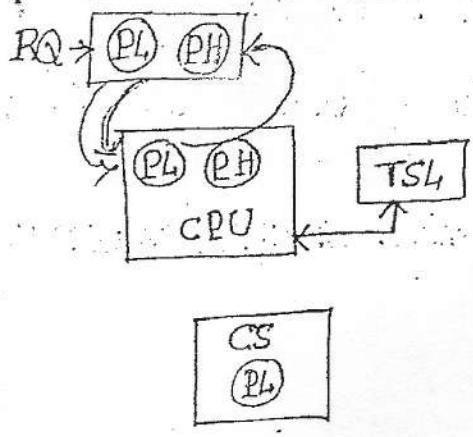
* Busy waiting => Priority Inversion problem.
* Deadlock => OS uses preemptive priority based schedule.

E: TSL and peterson solution suffers from
- Priority inversion
- Deadlock.

Priority Inversion:

* suppose in RQ, Process PL is arriving. it is having low priority.
* PL enters the CS by setting TSL register.
* Now a higher priority process comes into RQ (PH).
* Scheduler will preempt PL to take up PH.
* But PH cannot enter the CS as the TSL register value is set to 1 while PL was preempted. it was in the CS.
* Now PL will never get a chance to complete its CS. & set the flag value to '0'. as it will never be taken up by CPU, until higher priority Process PH is finished.

PH is in busy waiting condition, so PH will also never come out.



PH > PL => Deadlock.

* it will leads to a Deadlock.
* solution: Round Robin Scheduling.

Disabling Interrupt:

MUTUAL EXCLUSION WITHOUT BUSY WAITING: (Blocking)

ME without busy waiting using sleep() & wakeup().

Producer - consumer prob with sleep()/wakeup();

Producer:

```
Void producer()
{
   int itemp, in=0;
   while(1)
   {
   1. Produce item(itemp)
   2. if(count==n);
                  sleep() => if Buffer = FULL
                                 Producer => sleep().
   3. Buffer[in] = itemp;
   4. in = (in+1) mod n;
   5. count = count+1;
   6. if(count==1) ;
                  wakeup(consumer)
   }
}
```

consumer:

```
Void consumer()
  {
     int itemc, out=0;
     while(1)
     {
      1. if(count==0); sleep();
      3. out=(out+1) mod n;
      4. count= count-1;
      5. if(count == n-1); wakeup(Producer)
      6. consume item(itemc)
      2. itemc = Buffer[out]
  } }
```

RQ:→ Ⓟ Ⓒ N=3

CPU

X	0
Y	1
Z	2

0 1 2 3

Buffer (0-2)

Ⓒ : ..... → Preemption sleep.

Ⓟ : X
Y
Z

Inconsistency

sleep(); } Deadlock
Ⓒ : sleep(); }

* It has problem of inconsistency.
* suffers from potential Deadlock condition, due to lack of synchronization.

Explanation:

* Suppose, the consumer checks the value of count, before it can go to sleep, on seeing count==0, it gets preempted, Thus next time the first execution will take consumer to sleep() without further checking of count value.
* Now producer gets a change to run, it produced n items, and goes to sleep() on seeing count==n, after producing n items, notice the wakeup() call to consumer from producer consumer never got the change to go to sleep.
* Now consumer gets a chance to run, and it goes to sleep, right away, on account of its previous knowledge of count.
* Now buffer[] is FULL; both producer and consumer is in sleep(); waiting for being awaked by the each other

Deadlock

SEMAPHORES (Dijkstra's condition variable)

* semaphore is a process synchronization tool.
* semaphore is a variable that takes an integrated value.
* semaphore is a class with attributes: semaphore value.

  Functions: init(), wait(), signal().
* it is an operating system resource.
* semaphores can be used to implement Mutual Exclusion among any num. of processes.
* semaphore operations are executed in Kernal mode automatically (no preemption).
* There are 2 types of semaphores.
  1. counting semaphores (General semaphores)
  2. Binary semaphores (Mutuexes variable).

COUNTING SEMAPHORES: integer value can range over an unrestricted domain (-∞ to +∞)
counting semaphores are general semaphores.

BINARY SEMAPHORES: integer value can range only between 0 and 1; can be simpler to implement.

NOTE: counting semaphores can implemented by using binary semaphore.

semaphore operations:

DOWN	UP
P	V
wait	signal
↓	↓
decrement	increment.

COUNTING SEMAPHORE:

```
struct semaphore
{
    int value;
    Queue type L;
}
// List of those processes that get blocked while doing down operation, unsuccessfully.
```

I semaphore S;

II S.value = 4

III Down(S);

```
Down(semaphore S)
{
1. S.value = S.value - 1;
2. if (S.value < 0) // semaphore value becomes '-ve' and block it.
   {
3.   Put process(PCB) in SL
4.   sleep();
   }
5. else
     return;
}
```

* The +ve value of semaphore indicates the no.of down operations can be performed successfully.
* The -ve value indicates number of blocked processes.
* In critical section problem the value of semaphore is 1, as it indicates mutual exclusion; i.e only one process can enter the critical section, otherwise gets blocked.

NOTE: After executing the signal() operation, if the value of the semaphore is one or more than one, it means, there are no blocked processes.

* After executing the signal() operation, if the value of the semaphore variable is still negative (<0), it means there are still more than one blocked process.

* semaphore value is zero, then there is exactly one blocked process.

```
up(semaphore s)
{
 1. s.value = s.value+1;
 2. if(s.value<=0)
    {
     3. select a process from SL
     4. wake up();
```

↳ it means, the selected process can enter the critical section.

Eg: s.value = 4

4-1 = 3
3-1 = 2
2-1 = 1
1-1 = 0
} 4 successful processes available in stack.

0-1 = -1
-1-1 = -2
} 2 - blocked processes

⇓

unsuccessful processes

⇓

waiting for semaphore.

Eg: In a certain application the value of counting semaphore = 9; then the following operations performed on s

12P, 2V, 8P, 4V, 6P, 1V, 2P.

then what will be the current value of s=?

9 −12 +2 −8 +4 −6 +1 −2 = −12
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
S 12P 2V 8P 4V 6P 1V 2P

Eg: S = 12, 16P, 2V, 8P, 1V, 4P, 3V, 6P, 2V

S = 12 (16 down)
= −4 (2 up)
= −2 (8 P)
= −10 (1V)
= −9 (4P)
= −13 (3V)
= −10 (6P)
= −16 (2V)
= −14

P : down
V : up

Eg: At a particular time, the value of a counting semaphore is 10, it will become 7 after.

A. 3V
(B). 3P
C. 5V & 2P
(D). 13P & 10V

S = 10 (3P)
= 10 − 3
= 7

S = 10 (13P)
= (10 − 13)
= −3 (10V)
= 7

Eg: At a particular time of computation, the value of a counting semaphore is 7. Then 20P operations and 'x' V operations were completed on this semaphore. The final value of the semaphore is 5, x will be.

A. 22 (B). 18 C. 15. D. 13

S = 7 (20P)
⇒ −13 + x = 5
⇒ x = 18.

Eg: Mutual exclusion problem occurs between:

A. Two disjoint processes that do not interact.
(B) processes that share resource
C. Processes that do not use the same resource.
D. None.

Binary semaphore:

```
struct Bsemaphore
{
    enum value (0,1);
    Queue type L;
} // List of all PCB's whose corresponding
  Processes get blocked while performing 'down
  operation' unsuccessfully.
```

```
1. Bsemphore s;
2. s.value = 1;
3. Down (s);
```

```
Down (Bsemaphore s)
{
1. if (s.value == 1)
   {
     2. s.value = 0;
   }
   else
   {
     3. put the process in s.L and block it
     4. sleep();
   }
}
```

* if s.value = 1 down operation is possible change value from 1 to 0. operation Down() successful. process can enter CS.
* if s.value = 0, down operation is unsuccessful, process gets blocked.

→ In binary semaphore, we can not determine the no. Processes in the blocked queue by examining its value.

-> when the value of Bsemaphore = 0, then there might or might not be any blocked process.

-> By looking at the blocked queue status, we can come to know whether the queue is empty or any process is there in the queue.

-> As long as a process is in critical section or one or more process is in the block queue, the value of binary semaphore is zero. (0).

-> only processes have might to change the value of S to 1. if there is no process in blocked queue the process coming out of the CS, will make semaphore value to 1 from 0.

* up (Bsemaphore S)

```
{
  1. if (SL() is empty)
     {
       2. S.value = 1;
     }
  else
     {
       3. select a process from SL (only one process)
       4. wakeup();
     }
}
```

Eg: if S is a Binary semaphore and its value is 1 then what will be the value of S. after performing the following operation in order. 1P, 3V, 8P, 1V, 6P, 4V. & also what is the current value of L.length().?

S = 1, next after the 1st P, S = 0.

and Down(s): successes.

Now for the coming 11P next, S does not change (S=0), but all the 11 new process are put in the blocked Queue.

∴ L.length = 11.

→ Next 3V are attempted. All are unsuccessful. Q not empty ⇒ S=0, But 3 processes are moved out of L

∴ L. length = (11-3) = 8.

→ Next 8P are attemped. All are unsuccessful.

S=0, 8 more processes are added to the Queue.

∴ L. length = (8+8) = 16.

→ Next 1V is attemped, unsuccessful as Q is not empty.

S=0, one process is taken out of the blocked Queue

∴ L.length = (16-1) = 15.

→ Next, 6P are attempted, All are unsuccessful

S=0, 6 new processes are added to the blocked Queue.

L. length = (15+6) = 21.

→ Next 4V are attempted, all are unsuccessful as Queue is still not empty.

S=0, 4 more processes are moved out of the Queue

∴ L. length = (21-4) = 17.

Hence, after all those operations, value of the binary semaphore is: S=0.

∴ Length of the blocked Queue is L. length = 17.

4 cases of Binary semaphore:

case: I: $S=1$, $P(S) \Rightarrow S=0 \rightarrow P(S)$ is successful.

case: II: $S=1$, $V(S) \Rightarrow S=1 \rightarrow V(S)$ is successful

case: III: $S=0$, $P(S) \Rightarrow S=0 \rightarrow P(S)$ is unsuccessful.

case: IV: $S=0$, $V(S) \Rightarrow S=1 \rightarrow V(S)$ is successful.

=

eg: According to the following code, how many process can execute in the critical section.

```
mutex = 1;
Pi = 1,.. 9
while (1)
{
→ Down (mutex)
      <CS>
    up (mutex);
}
```

```
P = 10
while (1)
{
→ up (mutex)
    <CS>
  up (mutex);
}
```

Initially mutex is 1.

P_1 makes it '0' and enter into the critical section.

P_{10} is having an infinite loop, inwhich, it is performing up() before entering the CS as well as before coming out of CS.

* Because of the up() operation, before <CS>, it will be easy to follow that every time P_{10} enters <CS>, it make mutex = 1, instead of 0, indicating no process is running in the <CS> currently.

As a result processes, P_1 to P_9, it waiting in the Queue can perform the down() operation successful

⇒ At a time 10 processes can enter the <CS>.

```
P0()
{
while(1)
  {
    down(S0);
    Print(0);
    up(S1);
    up(S2);
  }
}
```

```
P1()
{
  down(S1);
  up(S0);
}
```

```
P2()
{
  down(S2);
  up(S0);
}
```

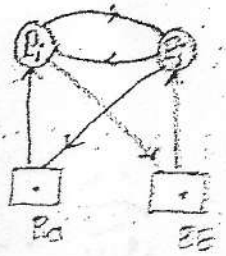
How many times '0' will be printed.

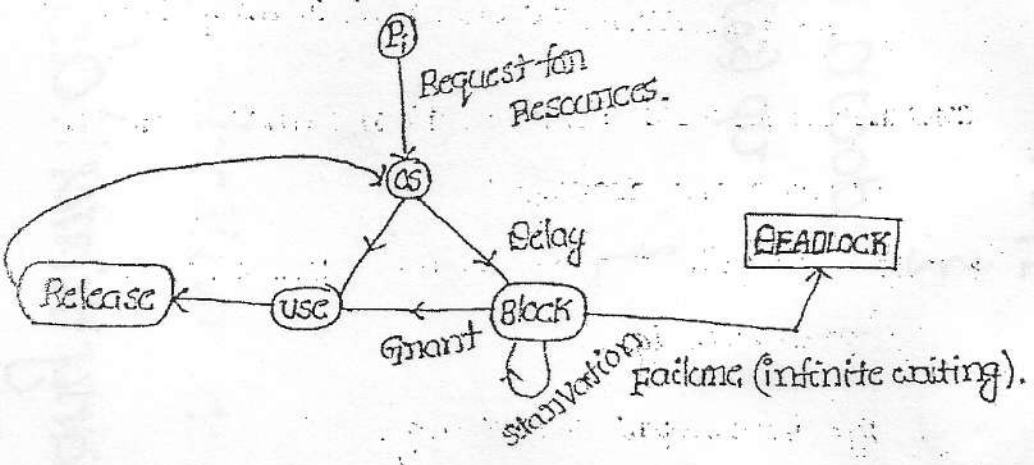
DEADLOCKS (undesirable resources of OS)

* Deadlock is a significant problem that can arise in a communication of cooperating or competing processes. (shared Resource)
* Two or more processes are said to be in deadlock iff they wait for happing of an event, caused by the otherones will never occur.

System Model:

- 'n' processes ($P_1 \dots P_n$)
- 'm' resources ($R_1 \dots R_m$)





Processes, make a request for resources, if the resource are available, then it is granted by os. if it takes some time for resource allocation, then process gets blocked, Blocked process requests for the resources after certain time, and if again not granted then the process moves to deadlock.

NOTE: * Starvation => Long waiting
* Deadlock => Infinite waiting.

NOTE: * Every starvation is not Deadlock.
* Every deadlock is a starvation.

⇒ Finite num of resources is available in the system. These resources are distributed among a num of competing proces

* Generally 2 categories of resources can be distinguished

1. Reusable Resources.
2. consumable Resources.

Reusable Resources: Reusable resource is one that can be safely used by only one process at a time.

* processes obtain resource units that they later release for reusage by other processes.

Eg: processors, I/o channels, Memory, Files, DB, Semaphor

consumable Resources: consumable resource is one that can be created and destoryed.

* There is no limit on the num of consumable resources of a particular type.

Eg: interrupts, signals, msg.

E: * A process must request a resource before using it & must release the resource after using it.

** The num of resources required may not exceed the total num of resources available in the sys.

Eg: if the system have 4 Printers then the request for Printers is equal or less than 4.

⇒ A Process may utilize a resource in only one of the following

* Request
* use
* Release.

Necessary conditions for occuring Deadlocks:

→ Mutual Exclusion
→ Hold and wait or partial allocation
→ No preemption
→ circular wait

Mutual Exclusion:

* only one process may use a resource at a time.
* once a process has been allocated to a particular resource it has exclusive use of the resource.
* No other process can use a resource while it is allocated to a process.
* If another process requests that resource, the requesting process must be delayed until the resource has been rele.

Hold and wait:

* Every hold and wait should not be deadlock.
* A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
* If deadlock ⇒ must be hold and wait process.

No-preemption:

* A resource can be released only voluntarily by the proce holding it, after that process has completed its task.

circular waiting:

There exists a set of waiting Processes $\{P_0, P_1 \ldots P_n\}$ such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2 \ldots, P_n$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource Allocation Graph: (RAG) (Multigraph):

* Resource Allocation Graph is used to describe the deadlock.
* RAG is also called as "system resource-allocation graph."
* Directed graph that depicts a state of the system of resource and processes.
* Graph consists of a set of vertices (V) and set of edges (E).

Processes: $P = \{P_1, P_2, P_3 \dots P_n\}$

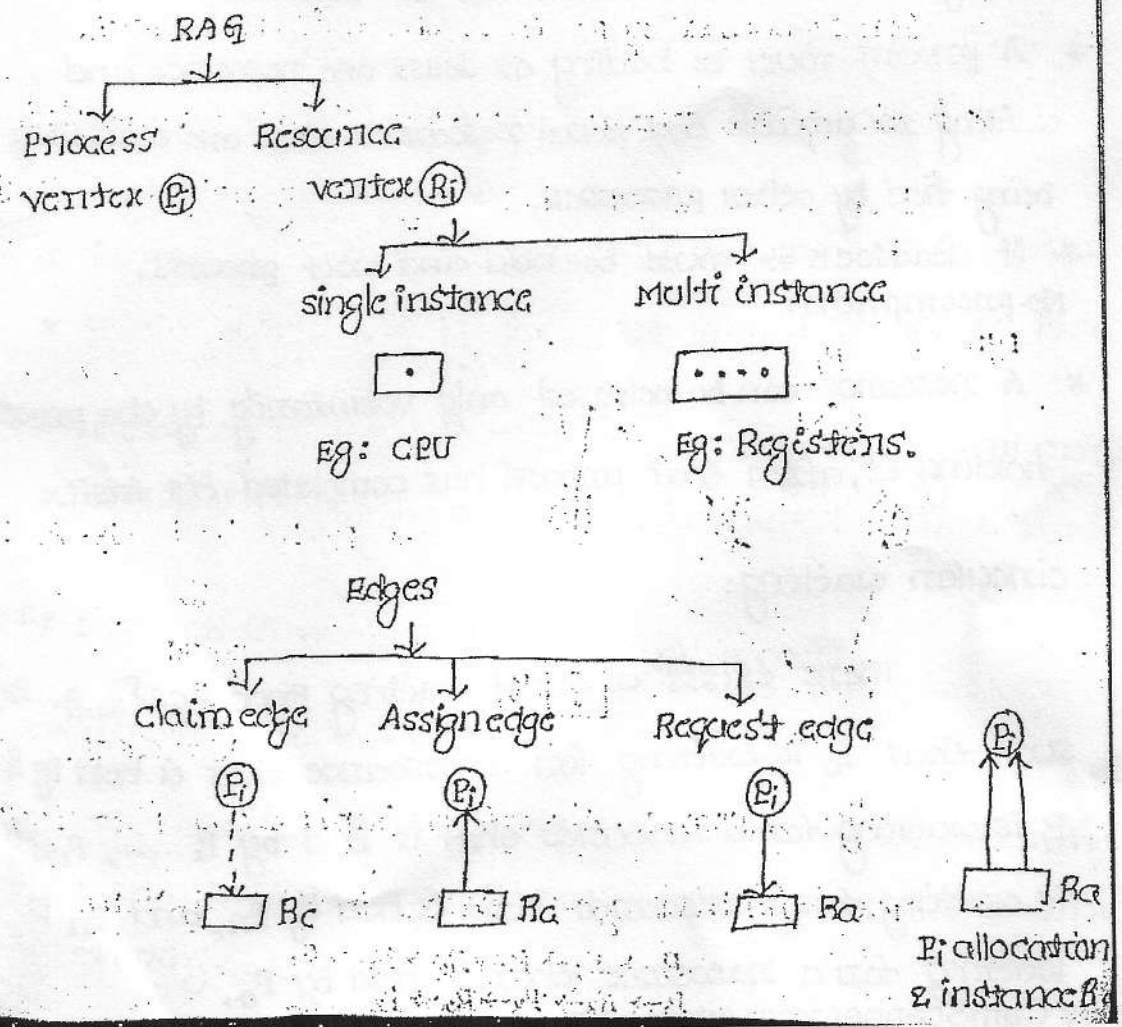
Resources: $R = \{R_1, R_2, R_3 \dots R_n\}$.

Request edge: Requesting of resource by Process.

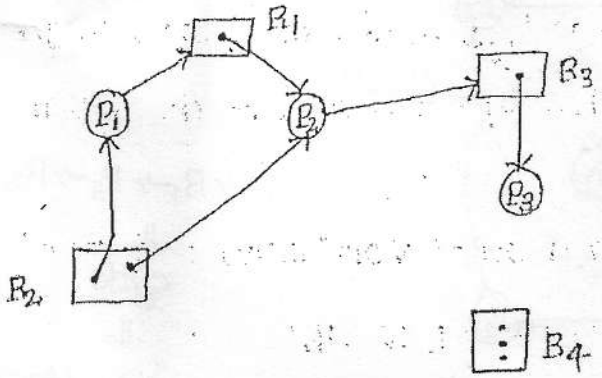
An edge from process to Resource => $P_i \rightarrow R_j$.

Assignment edge: Holding of resource by process.

An edge from Resource to process => $R_j \rightarrow P_i$.



* system consists of three processes i.e P_1, P_2, P_3, and 4 Resources i.e R_1, R_2, R_3, R_4



$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

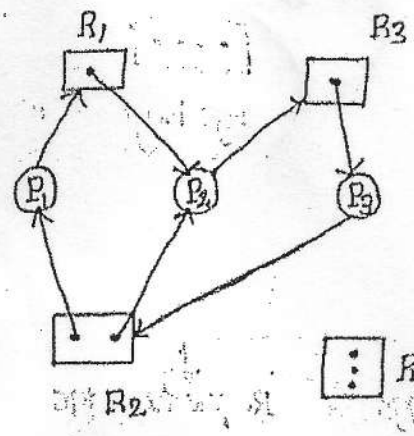
$E = \{R_1 \to R_1,\ P_2 \to R_3,\ R_1 \to P_2,\ R_2 \to P_2,\ R_2 \to P_1,\ R_3 \to P_3\}$

E: * if the graph contains no cycle, then no process in the system is deadlock.

* if the graph contain cycle => iff there is only one instan per resource type then there is a deadlock.

* if the graph contain cycle => iff there is multiple instance per resource type then may or may not be deadlock sta

Eg:



=> $P_1 - R_1 - P_2 - R_3 - P_3 - R_2 - P_1$

⇓

cycle

⇓

Deadlock.

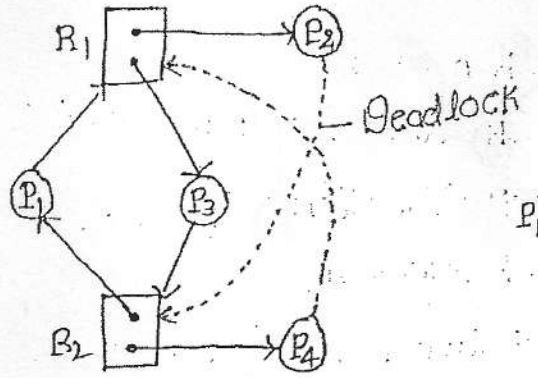
fig: Resource-Allocation Graph with Deadlock.

* two minimal cycles exists in the system.

$P_1 \to R_1 \to P_2 \to R_3 \to P_3 \to R_2 \to P_1$

$P_2 \to R_3 \to P_3 \to R_2 \to P_2$

eg: RAG with a cycle but no deadlock.



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2$

⇓

cycle

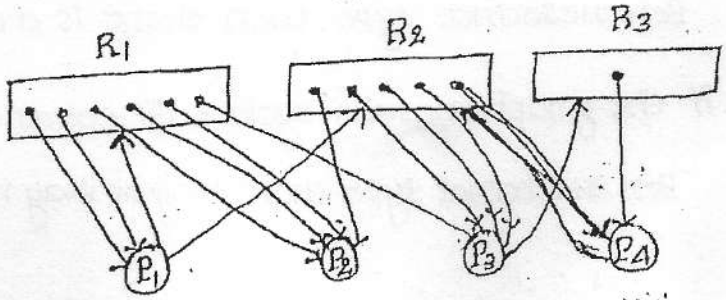
⇓

No Deadlock.

eg: Processes

	current Allocation			outstand Request		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	0	0	1	1	0
P_2	3	1	0	0	0	0
P_3	1	3	0	0	0	1
P_4	0	1	1	0	1	0

RAG:



Strategies for handling Deadlock:

1. Deadlock prevention } No Deadlock.
2. Deadlock Avoidance (Banker's Algo) }
3. Deadlock Detection } Deadlock occurs.
4. Deadlock Recovery }
5. Deadlock ignorance (ostrich algorithm).

1. Deadlock prevention:

"By dissatisifying one/more of the necessary condition"

* Deadlock prevention ensure that system will never enter in deadlock state.

Indirect Method: To prevent the occurance of the 3 necessary conditions ie Mutual Exclusion
Hold and wait
No-pricemption

Direct Method: To prevent the occurance of "Circular wait"

Mutual Exclusion: (Non-Dissatisifiable)

* Mutual Exclusion condition must hold for nonshareable resources.
* if access to a resource requires mutual exclusion then mutual exclusion must be supported by the OS.
* Holding only one process within the critical section.
* some resources such as files, may allow multiple access for reads but only exclusive acces for writes.
* Deadlock occurs if more than one process requires write permission.

Hold and wait : !(Hold & wait) = (Hold or wait)

* Hold and wait must guarantee that whichever a process requests a resource, it does not hold any other resources.
1. Requires a process must to request and be allocated all its resources before it begins the execution.
2. The Hold and wait conditions can be eliminated by forcing a process to release all resources held by it whenever it requests a resource that is not available.

(or)

"Release all the resources by force making a new request."

NO - pre-emption:

* If a process holding certain resources is denied a farthen request. That process must release its original resources and if necessary request them again together with additional resources.
* if a process requests a resource that is commently held by another process, the OS may preempt the second process and require it to release its resources.
* Sequential I/O devices cannot be preempted.
* pre-emption is possible for certain types of resources such as CPU and main memory.

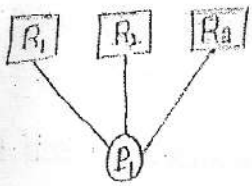
Circular wait:

* one way to prevent the circular-wait condition is by linear ordering of different types of sys resources.
* System resources are divided into different classes. if a Process has been allocated to resource type R, then it may subsequently request only those resource type following R in the ordering.

Example:

Mutual - Exclusion:

Holding only one process within the critical section.



I Phase	II Phase
R_1, R_2	R_2, R_3
30 min	40 min

A process P_1 needs to access the resources R_1, R_2 and R_3 if a resource R_1 and R_2 are accessed and another time again R_2 needs to be accessed.

Hold and wait:

* Process 'P' in I Phase requires R_1, R_2 resources and the time span allotted to phase-I is 30 min to complete.
* After the completion of Phase-I Process P release the resource R_1 but holds the resource R_2 and requests for R_3. This situation is considered as "Hold and wait".

Ist Approach:

Process in phase-I requests all the resources whatever it needs.

Eg: $P \rightarrow (R_1, R_2, R_3)$

They does not wait for all the resources to be accessed.

IInd Approach:

Release all the resources before making a new req

PHASE-I $\rightarrow P \rightarrow (R_1, R_2)$

PHASE-II $\rightarrow P \rightarrow (R_2, R_3)$

* After completion of phase-I release all the resources eventhough it needs same resources at phase-II.

* Again in phase-II, the resources are requested newly.

No-preemption:

Processes get preempted in two ways

- Forced preemption → (From others) → starvation, Inconsistency
- self preemption → (From resources) → starvation.

=

circular wait:

"Total order relation on all process for requesting of resource"

(or)

$R_1 - 8$
$R_2 - 7$
$R_3 - 6$
$R_4 - 4$
$R_5 - 12$

" Number all resources uniquely process must not request a lower num. of resource than the one last request earlier".

* Process P_i requests in any order, if no order is followed, any sequence is followed, P_i must release all resources.

DisAdv: starvation.

Advantages:

* No preemption is necessary:
* works well for process that perform a single burst of activity.
* Needs no run-time computation.

DisAdv:

* Inefficient
* Delay process initiation.
* Disallow incremental resource req.

=

1) Which of the following is NOT a valid Deadlock prevention Scheme.

A. Release all resources before requesting a new resource.

B. Number all resources uniquely and never a lowest numbered resource than the last one requested.

(C) Never request a resource after releasing any resource.

D. Request and be allocated all required resources before execution.

A computer has six Tape Drives. with n processes competing for them Each process may need Two drives. what is the max value of 'n' for the sys to be Deadlock free.

(A) A. 6
(B) 5
C. 4
D. 3

$n = 5$
$m = 2$
$R = 6$
$P_1 \rightarrow 2(R)$

$P_1 - 1$
$P_2 - 1$
$P_3 - 1$
$P_4 - 1$
$P_5 - 1$

$\therefore n = 5$

Eg: Suppose there are 15 processes and each process requires 3 instance of resources type R, the min num. of resources required in order to ensure the execution of all the processes without deadlock.

A. 31
B. 44
C. 27
D. NONE

$R = n * (S-1) + 1$ (S: instance of resource type)

$R = 15 * (3-1) + 1$

$R = 31$

$R = ?$
$n = 15$
$S = 3$

Eg:

$R = 19$
$n = ?$
$S = 3$

$R = n * (S-1) + 1$

$19 = n * (3-1) + 1$

$2n = 18$

$n = 9$

(i) $n = 3$

$R =$

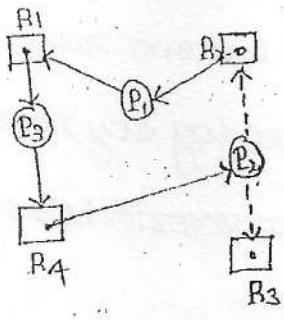
$S = 2$

$\Rightarrow R = n*(S-1)+1$

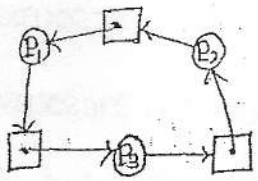
$= 3*(2-1)+1$

$R = 4$

(or) $\left.\begin{matrix} P_1 - 1 \\ P_2 - 1 \\ P_3 - 1 \end{matrix}\right\} 1 \Rightarrow R = 4$



NO DEADLOCK.



DEADLOCK.

(ii) $R = 6$

$n =$

$S = 3$

$\Rightarrow R = n*(S-1)+1$

$\Rightarrow 6 = n*(3-1)+1$

$2n = 5$

$n = 2$

Q. Consider a system having m resource of the same type. These resources are shared by 3 processes A, B and C, which have peak demands of 3, 4, and 6 respectively. For what value of m Deadlock will not occur.

A. 7
B. 9
C. 10
D. 11

Sol:

P	R_{max}
P_1	3 - 2
P_2	4 - 3
P_3	6 - 5
	10

$\Rightarrow$ No.of min resources with which deadlock will occur = (2+3+5)

= 10

∴ No.of min resources with which dead lock will not occur = 10+1

= 11.

Eg:

P	R_{max}	
P_1	10	- 9
P_2	18	- 17
P_3	3	- 2
P_4	15	- 14
P_5	6	- 5
		47

$\Rightarrow$ Min num.of resource to avoid deadlock

= 47+1

= 48.

2. Deadlock Avoidance (BANKER'S ALGORITHM):

Deadlock can be avoided by using

- → Banker's Algorithm
- → SAFE state Algorithm
- → Resource Request Algorithm.

objective: obtain a system always in a SAFE mode.

* it is an "Apriori information" algorithm, it is based on STATE of the system.

SAFE
UNSAFE → cunning of deadlock.

SAFE STATE:

if the need of all the processes can be satisfied with the free/available resources in some order or sequence then the system is said to be in SAFE STATE.

* if a system is in a SAFE STATE ⇒ NO DEADLOCK.

UNSAFE STATE:

if the system has allocated available all resources to all the processes's need, then it is in potentially UNSAFE.

* if a system in UNSAFE state, it may or may not be in deadlock
* Deadlock is a subset of UNSAFE state.

So, if the system is in deadlock, it is surely in UNSAFE STATE.

working principle of Banker's Algorithm:

Never allow the system to go to the UNSAFE STATE

BANKER'S ALGORITHM:

* Applicable for systems which is having multiple instances for each resource type.
* Each process must declare their peak demand beforehand.
* if a process requests a resource, it must wait, if required.
* Allocated resources must be given back, within a finite amount of time.

Data structure:

n → no.of Processes

m → no.of resources types.

(i) Available: A vector of length m. if Available[j] = K

⇒ There are K instances of resource type R_j currently available.

(ii). Max: A matrix of order n x m. if Max[i][j] = K

⇒ Process P_i may request atmost K instances of resource type R_j.

(iii). Allocation:

* An n x m matrix defines the num.of resources of each type currently allocated to each process.
* Allocation[i][j] = K

Process P_i currently allocated K instances of resources R_j

(iv). Need:

* An n x m matrix indicates the remaining resource need of each process.

Need[i][j] = K. ⇒ Process P_i still needs K instances of the resource R_j to complete.

SAFETY ALGORITHM:

1. Let 'work' and 'Finish' be vector of length 'm' and 'n' respectively.

   initialize : work = Available

   Finish = False ; ∀ i=0 to n-1.

   i.e work[i] = Available[i]

   Finish[i] = False.

2. Find an i such that : (A) Finish[i] = False

   (B) Need ≤ work[i]

   if no such 'i' is founds go to step 4.

3. work[i] = work[i] + Allocation[i]

   Finish[i] = TRUE

   go to step 2.

4. if Finish[i] == TRUE ∀i, system is in SAFE STATE.

   otherwise state is unsafe.

Resource - Request - Algorithm:

Request[i] = A vector of length m, request for process P_i

if Request[i][j] = K. => Process P_i wants K instances of resource type R_j.

1. Request ≤ Need go to step 2.

   otherwise raise an error condition since process has exceeded its maximum claim i.e Max[i].

2. Request[i] ≤ Available[i] go to step 3.

   otherwise P_i has wait since resources are not available.

3 (a). Available [i] = Available[i] - Request[i]

(B). Allocation[i] = Allocation[i] + Request[i]

(C) Need[i] = Need[i] - Request[i]

4. if SAFE => Resources are allocated to P_i.

5. if UNSAFE => P_i must wait for the Request[i] and the old resource-allocation state is restored.

=

Eg: System consists of 5 Processes (P_1, P_2, P_3, P_4, P_5) and three resources (R_1, R_2, R_3). Resources R_1 has 10 instance, Resource R_2 has 5 instance and Resource R_3 has 7 instances.

Process	Alloc	Max	Need	Available
P_1	0 1 0	7 5 3	7 4 3	3 3 2
P_2	2 0 0	3 2 2	1 2 2	5 3 2
P_3	3 0 2	9 0 2	6 0 0	7 4 3
P_4	2 1 1	2 2 2	0 1 1	7 4 5
P_5	0 0 2	4 3 3	4 3 1	7 5 5
				10 5 7

SAFE => $< P_2, P_4, P_5, P_1, P_3 >$

The new system states as.

Process	Alloc	Need	Available
P_0	0 1 0	7 4 3	2 3 0
P_1	[3 0 2]	[0 2 0]	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 2 2	4 3 1	

Now, we run the safety algorithm

work = Available = (2,3,0)

we have for P_1 : Finish [i] ≠ TRUE

Need [i] ≤ (0 2 0) ≤ (2,3 0)

$\Rightarrow$ work = work + Allocation [1] = (5 3 2)

Finish[1] = TRUE

For P_3, we have

Finish[3] ≠ TRUE

Need[3] = (6,1,1) ≤ (5,3,2)

$\Rightarrow$ work = work + Allocation [3] = (7,4,3)

Finish [3] = TRUE.

For P_4 we have,

Finish [4] ≠ TRUE

Need[4] = (4,3,1) ≤ (7,4,3)

$\Rightarrow$ work = work + Allocation[4] = (7,4,5)

Finish [4] = TRUE.

For P_2 we have

Finish [2] ≠ TRUE

Need[2] = (6,0,0) ≤ (7,4,5)

$\Rightarrow$ work = work + Allocation[2] = (10,4,5)

Finish[2] = TRUE.

For P_0 we have,

Finish[0] ≠ TRUE

Need[0] = (7,4,3) ≤ (10,4,5)

$\Rightarrow$ work = work + Allocation[0] = (10,5,7)

Finish[0] = TRUE.

∴ The system is in SAFE Sequence: < P_1, P_3, P_4, P_2, P_0 >

For P_4 : Request[4] = (3,3,0) can not be granted

as (3,3,0) ≰ (2,3,0) = available

3. DEADLOCK DETECTION:

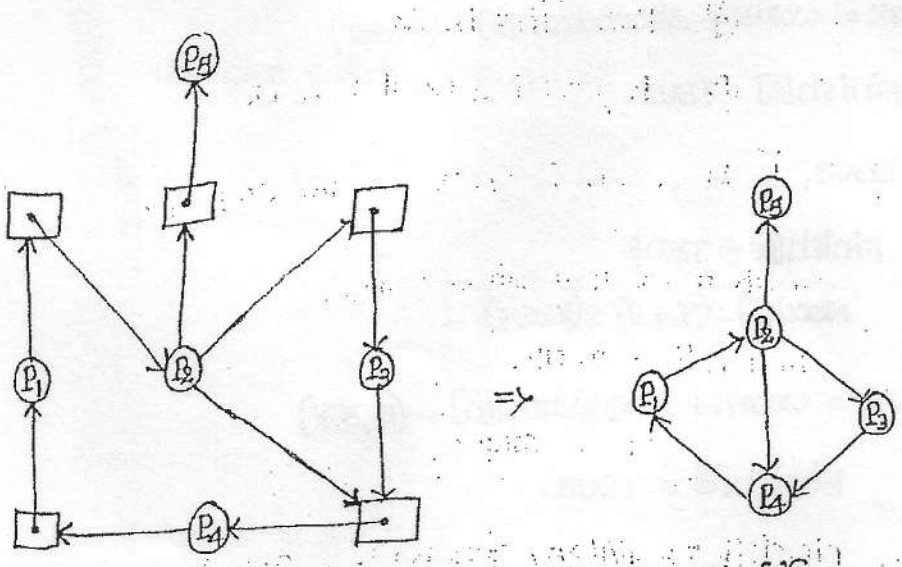
* Deadlock detection approach do not limit resource access or restrict process actions.
* with deadlock detection request resources are granted to Processes whenever possible.
* The OS Performs an algorithm that allows it to detect the circular wait condition.

single instance of each Resource type:

(WAIT-FOR-GRAPH)

* if all resources have only a single instance, then deadlock detection algorithm uses a variant of the resource allocation graph called as "wait-for-graph".
* wait-for-graph is obtained from Resource-Allocation graph.

RAG → WG => Remove the resource nodes and collapsing the appropriate edges.



RAG

WG

=> $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1$

⇓

cycle

⇓

Deadlock.

System with multiple instances of resource:

RAG as well as WG is not good to detect deadlock, we can use the following algorithm which is allmost identical to SAFETY ALGOB.

New data structure:

Request[i][j]: Request is a m×n matrix.

Request[i][j] = K, => Procen P_i has requested for K instance of resource type R_j.

Algorithm:

1. Let work and Finish be vecton of lengeh m and n respectively.

initialize: work = Available

Finish[i] = False, if allocation[i] ≠ 0 and

Finish[i] = TRUE otherwise.

2. Find an index i such that both

(A). Finish[i] = FALSE.

(B). Request[i] ≤ work; holds.

if no such i exists go to step 4.

3. work = work + Allocation[i]

Finish[i] = TRUE.

Go to step 2.

4. Finish[i] == FALSE, for some i $0 \le i \le n$, => the system is in a deadlock state.

Moreover Finish[i] == FALSE, then procen P_i is in deadlock.

Find if there is a deadlock, or not, in the following system when process P_2 makes an additional request for an instance of type C.

$n = 5$ $(P_0 - P_4)$

ABC = (7 2 6)

Process	Alloc	Request	Available
P_0	010	000	000
P_1	200	202	010
P_2	303	000	313
P_3	211	100	524
P_4	002	002	526
			726

SAFE STATE : $< P_0, P_2, P_3, P_4, P_1 >$

Upon receiving a new request from P_2 as : Request[2] = (0 0 1)

The new request table constructed as follow:

Process	Alloc	Request	Available
P_0	010	000	000
P_1	200	202	010
P_2	303	001	
P_3	211	100	
P_4	002	002	

$< P_0 \quad >$ => Deadlock

=> P_1, P_2, P_3, P_4 are all involved in a deadlock

=

(100)

$n = 3$

$(X\ Y\ Z) = (5, 5, 5)$

=> 555
543
012

Processes	Alloc X Y Z	Req	Avail
P_0	121	103	012
P_1	201	012	213
P_2	221	120	334
			555

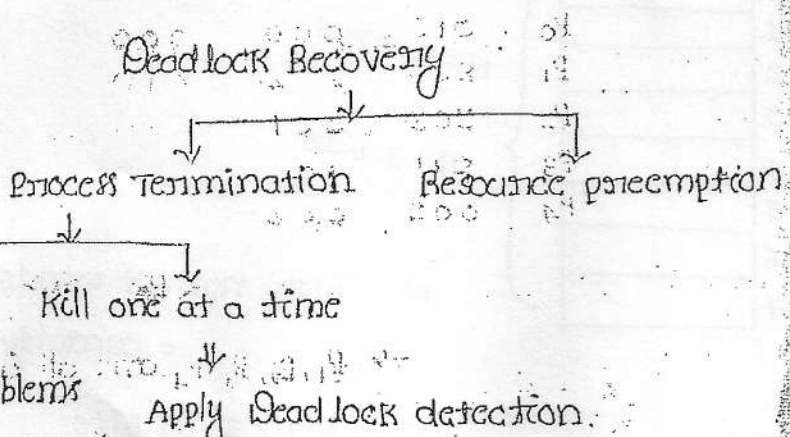
$< P_1\ P_0\ P_2 >$

∴ LAST finish process : P_2

4. Deadlock Recovery:

once deadlock has been detected; some strategy is needed for recovery.

1. All deadlock processes are aborted. Most of the OS apply this solution.
2. Abort one process at the time until the deadlock no longer exists, The order in which processes are selected for abortion should be on the basis of some criterion of min cost.
3. Successively preempt resources until deadlock no longer exists.
4. Backup each deadlocked process to some previously defined check point and restart all processes. This required roll back and restart mech are build in to the sys.

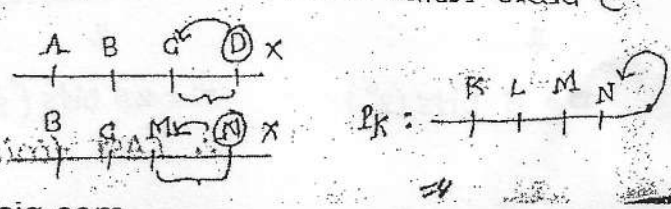


=> No need to apply Deadlock detection.

Resource preemption:

Roll back

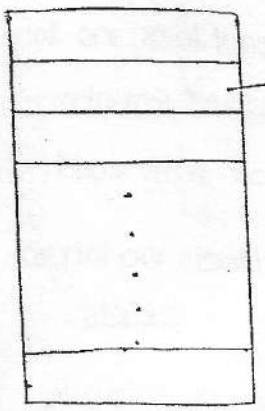
undo the work a process has done on last resource Roll back the process to the last known safer state & restart it.



MEMORY MANAGEMENT.

Abstract View of Memory:

1-D Array of words or linear one-dimentional Array



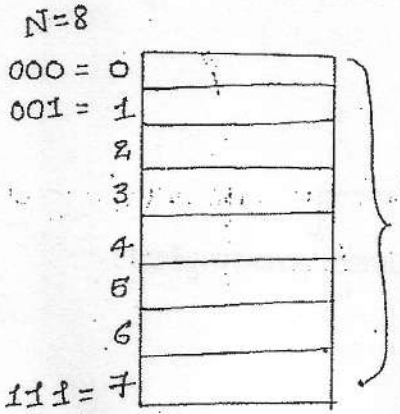
→ words/location/memory cells.

⇓

contains either data or instru

Each word is associated with address and word length.

NOTE: Total num.of set of words = Address space.



N = Total num.of words (Bits/Bytes)
= capacity of Memory.

n = Address in bits

m = Width/size/length of words (Bits)

NOTE: $N = 2^n$ → Num.of words. $n \propto N$.

$n = \log_2^N$ → Num.of bits

Eg: N = 64 words ⇓ n = 6 bits (2^6)

N = 512 M words ⇓ n = 29 bits ($2^9 \cdot 2^{20}$)

N = 4 G words ⇓ n = 32 bits ($2^2 \cdot 2^{30}$)

$2^5 = 32W$

$2^6 = 64W$

$2^7 = 128W$

$2^8 = 256W$

$2^9 = 512W$

$2^{10} = 1024W \cong 10^3 \cong 1K$

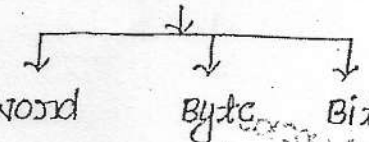
$2^{20} \cong 1024 * 1024 \cong 10^6 = 1M$

$2^{30} = 10^9 = 1G$

$2^{40} = 10^{12} \cong 1T$

$2^{50} = 10^{15} \cong 1P.$

Memory specification



* Memory capacity and addressability at the level of words, Bytes and bits.

Eg: n = 23 bits

m = 16-bits (1word = 2Bytes)

	capacity Address
Words	8MW 23-bits => 8MW $(2^3 \cdot 2^{20}) = 2^{23} \Rightarrow 23$
Bytes	8M*2B = 16MB 24-bits
Bits.	16M*8 bits 27 bits = 128Mbit

Eg:

	capacity	Address
words	256 MW	28-bits
Bytes	256 M * 4 B = 1GB	30-bits
Bits	1G * 8 bits	33-bits

N = 256 MW
m = 32-bit (1word = 4B)

Eg: N = 256 G bits
m = 64 bits (1 word = 8 bytes)

	capacity	Address
words	4 GW	32 bits
Bytes	4G * 8B	35 bits
Bits	256 G bits	38 bits

$\Rightarrow 256\,G * 8B$
$\Rightarrow 2^8 \cdot 2^{30}$
$\Rightarrow \frac{2^{38}}{2^6} = 2^{32} = 4\,GW$

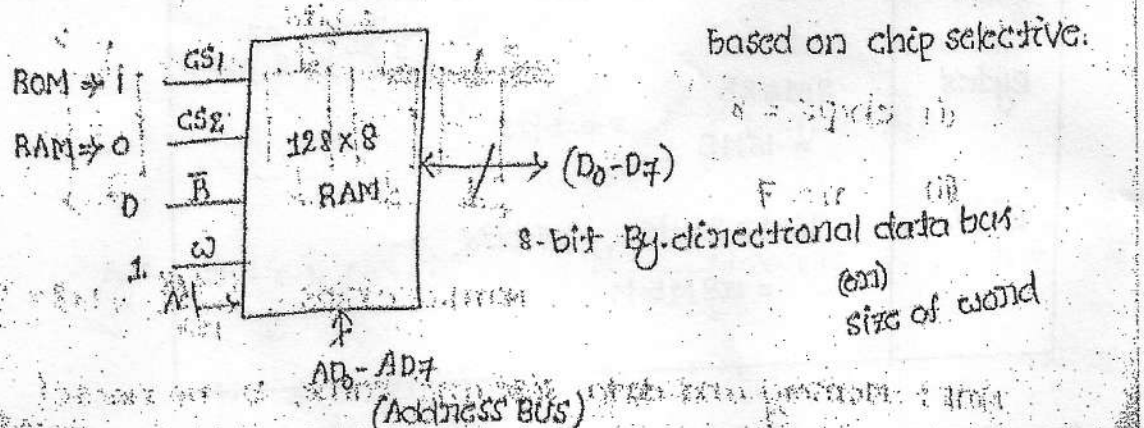
Eg: N = 512 G bits
m = 64 bits (1 word = 8 Bytes)

	capacity	Address
Words	8 GW	n = 33-bits
Bytes	8G * 8B = 64 GB	n = 36-bits
Bits	512 G bits	n = 39 bits

$\Rightarrow N = 512\,G\,bits = 2^9 \cdot 2^{30} = 2^{39}$
$\Rightarrow W = \frac{2^{39}}{2^6} = 2^{33} = 8\,GW$
n = 33-bits

RAM chip Organization:

NOTE: Read of data from RAM or ROM based on chip selective.



Number of chips required = $\frac{\text{Desired Memory size}}{\text{Memory chip size.}}$

Eg: Memory size = 512 MB

chip size = 128 MB

$\therefore$ Num. of chips = $\frac{512\ MB}{128\ MB} = \frac{2^9}{2^7} = 2^2$

$= 4$

Eg: Memory size = 16 KB (Vertical)

RAM = 128 × 8

↳ chip size = 128 words $= 2^7$

Memory size = 16 KB $= 2^{14}$ B $\Rightarrow$ Here (1 word = 8 bits)

(i) Num. of chips = $2^{14}/2^7 = 2^7 = 128$ chips

(ii) n = 14-bits

(iii). Decoder = 7 * 128.

The decoder assembly is needed here.

$\Rightarrow$ CPU point of view - Address length = 14 bits = 16 KB

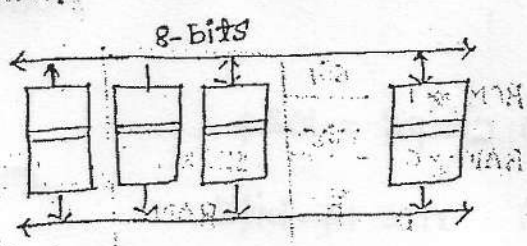
$= 2^{14}$ B $= 2^{14}$ W.

For each RAM chip Address length = 7 bits

$= 128 \times 8 \Rightarrow 2^7$ W $= 2^7$ B.

Eg: Memory size = 128 B (Horizontal)

RAM = 128 × 1 B $\Rightarrow$ 1 bit contains each word = 1-bit

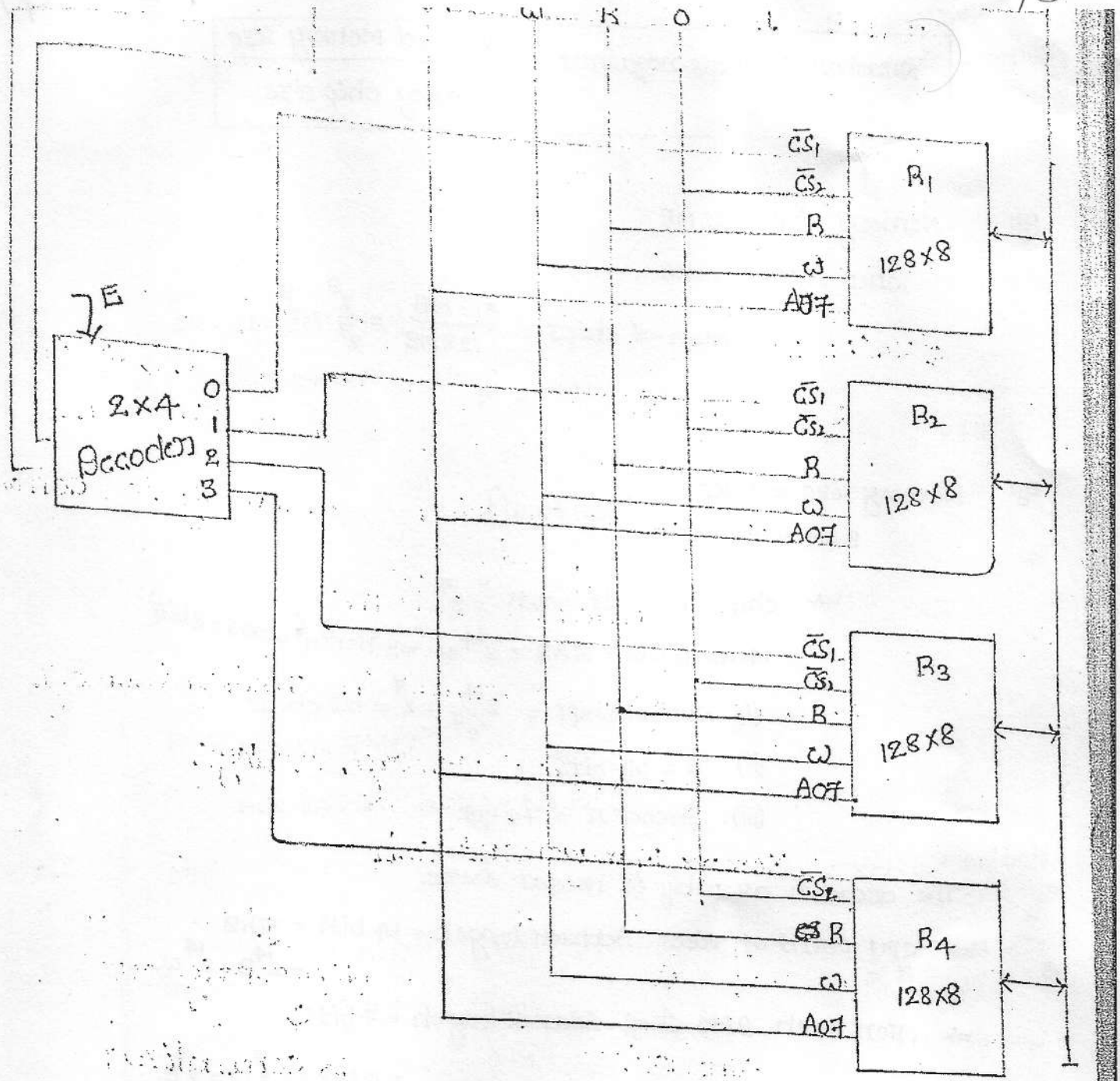


(i) chips = 8

(ii) n = 7

Num. of chips = $\frac{128 \times 8}{128} = 1 \times 8 = 8$

NOTE; Memory and data size are same, so no need of decoder.



Eg:

Memory size = 16 KB (Both vertical & Horizontal)

RAM = 128×1

$$\text{Num. of chips} = \frac{16\text{KB}}{128\times1} = \frac{2^4 \cdot 2^{10}}{2^7}\text{B}$$

$$= 2^7\text{B} = 2^7 \cdot 2^3$$

$$= 128\times8$$

$$= 1024$$

(i) chips = 1024

(ii) n = 14-bits

(iii) Decoder = [redacted] 10×1024

: Memory Management Techniques:

Functions:

=> Memory allocation
=> Memory Protection
=> Free space management
=> Deallocation.

Goals: 1. Effective utilization of memory space.

↳ Keeping fragmentation at each level, so there a chance of effective utilization.

=> Fragmentation => Memory loss can occur

Internal External

2. Run larger programs in smaller memory.

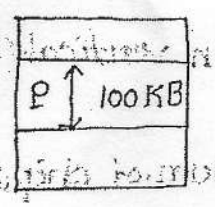
Eg: Virtual Memory, overlays.

=

Memory Management Tech

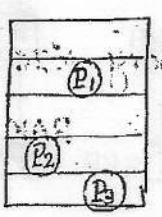
1. contiguous
2. Non-contiguous

contiguous ⇓ centralized



=> overlays
=> Partitioning
* Fixed partition
* variable partition
=> Buddy system.

Non-contiguous ⇓ Decentralized



* Paging
* segmentation
* segmentation with Paging
* Virtual Memory.

1. Overlays: Largest programs run in small memory space

Consider a 2-pass assembler.

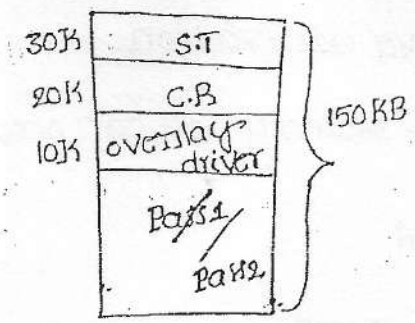
Pass 1 : 70 KB

Pass 2 : 80 KB

Symbol table : 30 KB

common Routine : 20 KB.

Total : 200 KB → Memory required.

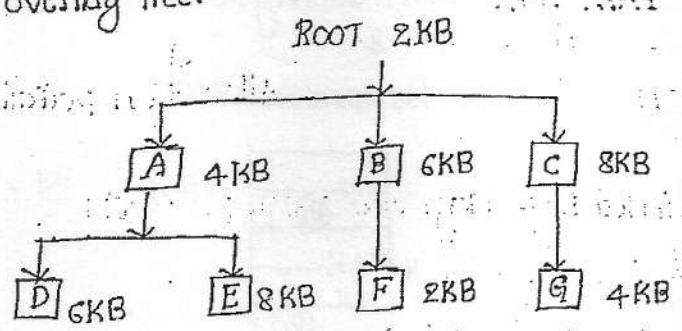


* For pass 1 or Pass 2 → sufficient memory is there.

* overlay driver → Pass 1 overwritten by Pass 2 and vice versa.

NOTE: Overlays are possible only when pass 1 and Pass 2 are independent to each other otherwise overlaying is NOT Possible.

Q5. Overlay Tree:



What will be the size of the partition required to load this Program.

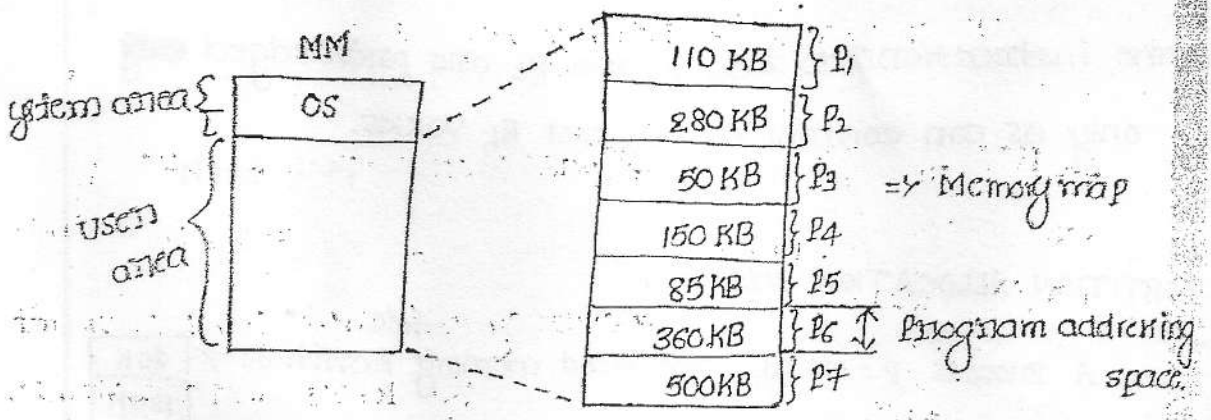
A. 12KB (B) 14 KB C. 10KB. D. 10KB.

→ if A, B, C are independent then maximum amount of memory required = 14 KB.

PARTITIONING

- Fixed partitioning
  - 1. Multiprogramming with fixed tasks (MFT)
- Variable partitioning
  - 2. Multiprogramming with variable tasks (MVT)

1. Fixed partitioning (MFT):

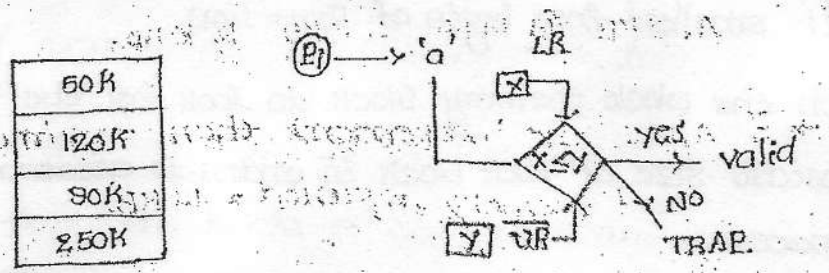


* user area can be divided into num. of fixed partition of different sizes.
* one partition according to one process

∴ 1 Partition = 1 Process.

PARTITION

- Protection
  - using pair of limit Reg + Magnitude comparator
    - Upper limit Reg → Fixed Address.
    - Lower limit Reg → First Address of partition.
- Allocation policies



* For each process a register called Base Reg holds its starting address. Another register called Limit Reg, holds the maximum memory limit that the process can use.
* For a process P_i if the Base Reg holds value R_B and Limit Reg holds value R_L, then the range of memory accessible by P_i is

$$R_B : (R_B + R_L)$$

* The instructions to load R_B and R_L are privileged only OS can control the R_B and R_L values.

PARTITION ALLOCATION POLICIES:

A Process P = 100KB The fixed memory Partition => free

80K
120K
200K
500K

then, where we can allocate the process by using Partition allocation policies.

1. First Fit (FF)
2. Best Fit (BF)
3. Next Fit (NF)
4. Worst Fit (WF)

First Fit: The first free begin enough to accomdate.

* The first block which is free is allocated to the new process (if it is longer enough to accommodate)
* Needs to scan the memory from begening.

Drawback: Internal fragmentation.

Best fit: smallest free begin of Partition.

* search the whole memory block to look for the most appropriate size of free block in order to accomodate the new process.

* Internal fragmentation is less.
* searching time is more.

Best available fit: Among available partitions select the smallest best available.

Best fit only: it selects for best fit only, it is not available it waits for that part partition and after available it takes.

Next fit:

* works like FF, but it starts the scanning of the memory from after the last allocated block instead of the very begining.

Adv: searching becomes faster than FF

worst fit:

* places the new processes in the largest memory block available.

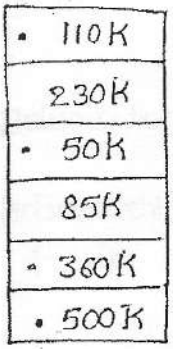
Performance of Multipule programming with fixed tasks:

(i) Fragmentation:

* Internal fragmentation: Amount memory can be wasted after the procer has been allocated.
* External fragmentation: when unable to accomodate a process of size even though we have sufficient size of memory because of contiguity tech.

Eg: If a new procer of size 300KB place into memory here the blocks of size 230K and 85 are free. The new Procer can not be allocate to it.

* An external fragmentation of total (230+85) = 315 K, will be in the memory and still the 300K size process has to wait.



→ contiguous memory allocation leads to External fragmentation.

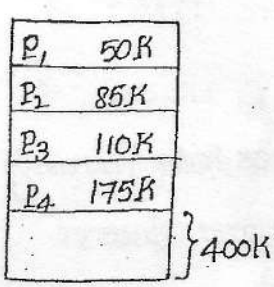
(ii). Maximum process size: Maximum partition size
⇓
No flexibility.

(iii). Maximum degree of multiprogramming is fixed.
→ Restricted no. of partitions. (can't not > n).

(iv). Best fit is superior to all because of low internal fragmen

2. VARIABLE / DYNAMIC PARTITIONING (MVT):

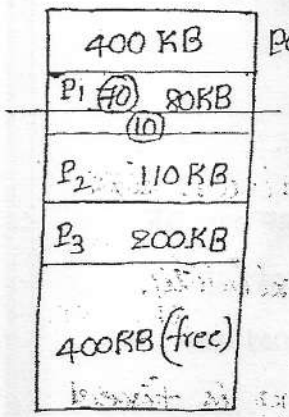
* Dynamic approach.
* No user area for partitioning.
* A partition control Manager (PCM) is used to create the need of incoming Process. if needed, it breaks down a large chunk of free memory space in order to accomodate a small process and also merge back the memory released after process has terminated into a large chunk.



Protection:

* impossible to use limit reg as logically infinite number of Partition variable size are possible.

* PCM is a seperate memory, is used to store the status, lower bound and upper bound of each partition allocated dynamically.



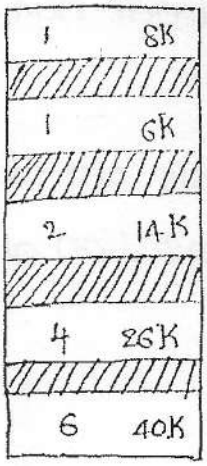
Eg: if a procck 70KB is needed to fit into memory, then if we take first fit, the 80KB partition is divided as 70KB for the process & rest of takes 10KB.

∴ we overcome fragmentation.

Performance MVT:

* No internal fragmentation
* External fragmentation is there.
* using dynamic relocation, procekes, shifted in main mem so that all the free/available blocks are put together and formed a large chunk, this way external frag is reduced.
* Degree of multiprogramming: no restriction to no. of Partition ⇓ flexible.
* Best allocation polices for this = " worst fit."
* "Best fit" : worst performance.

Eg: Memory Map



(a) $P_1 \rightarrow$ 6K

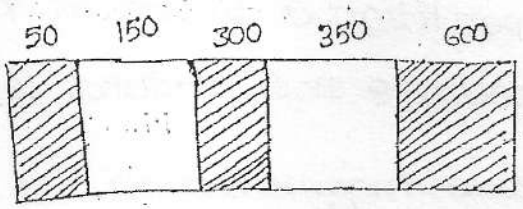
FF - 8K
BF - 6K
NF - 26K
WF - 40K

How many successive request max of size 6K can x satisfied.

(A). 5. (B). 14 (C). 13 (D). 12

Q18

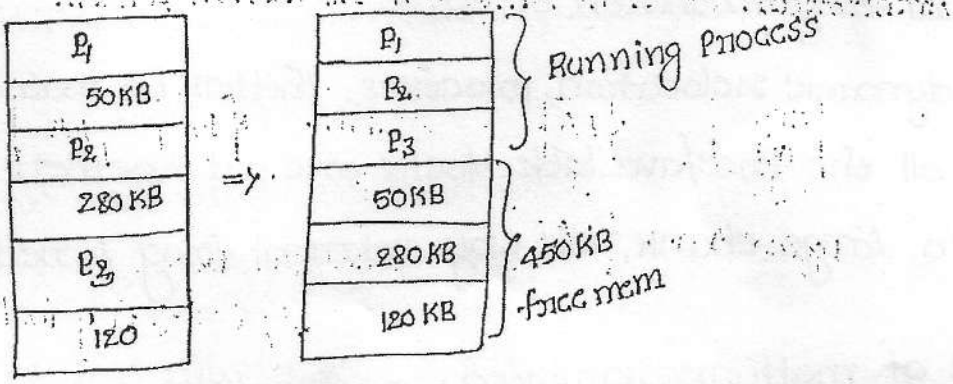
Eg: process requests: 300K, 25K, 125K, 50K.



(A) Either FF or BF
(B) FF but not BF
C. BF but not FF
D. None.

Solutions of External fragmentation:

(1) Defragmentation / compaction:



All the processes at different memory locations are compacted to one side & the free memory is now available.

* Process should not be dynamically relocatable and can process

Drawback: compaction is time consuming processes. It's a overhead to the OS, but it's desirable, but result in decreasing Performance.

(2) Non-contiguous Memory Allocation:

To overcome external fragmentation, we use non-contiguous allocation policies.

Address space: Space of words/series of words/set of location of memory

Address space

Virtual/Logical Address space

Physical Address space

Total amount of main memory

* Virtual/Logical Address space & Physical Address space:

Address bus: n-bit ⇒ LA

VAS/LAS = 2^n bits

LAS = 2^{LA}

LA = $\log_2^{(LAS)}$

4 GB RAM ⇒ PAS

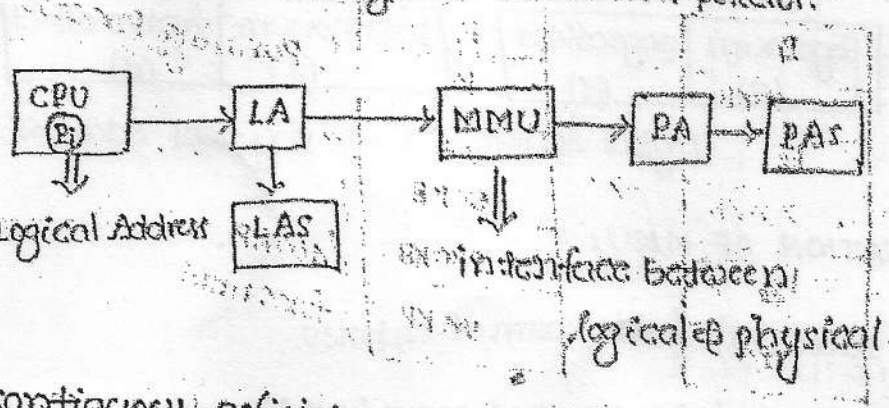
4GB = 2^{32} bit

PA = 32-bit

PAS = 2^{PA}

PA = $\log_2^{(PAS)}$

Architecture of non-contiguous Allocation policies:



Non-contiguous policies:

-> Organization of LAS/VAS

-> Organization of PAS

-> Organization of MMU

-> Translation process.

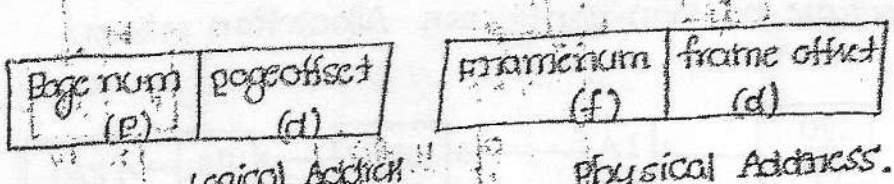
LAS > PSA

ORGANISATION OF LAS/VAS:

* LAS is divided into equal num of pages
* Page size is always power of 2. ($i.e\ 2^K, K>0$).
* Number of pages: $N = \frac{LAS}{PS} = \frac{2^{LA}}{2^K} = (2^{LA-K})$
* No of bits needed to locate all pages: $p = \log_2^N$ $i.e\ N = 2^p$
* page offset bit: $d = \log_2^{PS}$ $i.e\ PS = 2^d$
* Logical address bits: $(p+d)$ where $LAS = 2^{(p+d)}$

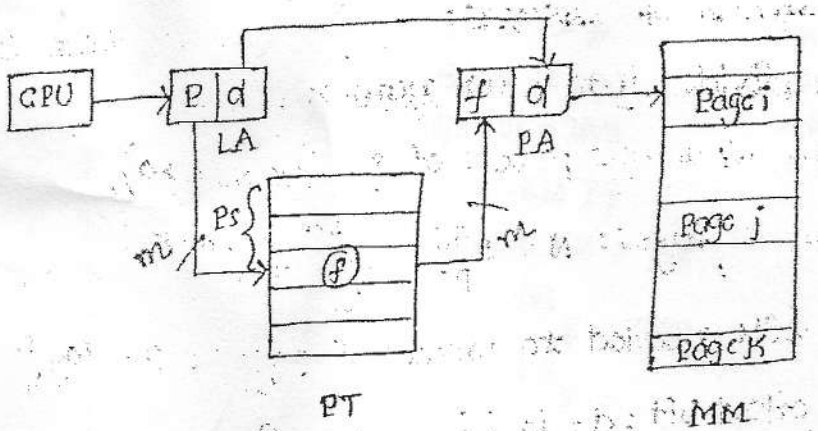
ORGANISATION OF PAS:

* PAS is divided into equal size of frames (blocks) [page frame]
* Frame size = page size
* Number of frames: $M = \frac{PAS}{FS} = \frac{2^{PA}}{2^K} = (2^{PA-K})$
* Num of bits needed to locate all frames: $f = \log_2^M$ $i.e\ M = 2^f$
* page offset (LAS) = frame offset (PAS) = d-bits $d = \log_2^{PS}$
* Physical Address = $(f+d)$ bits $\Rightarrow$ $PAS = 2^{(f+d)}$



ORGANISATION OF MMU:

* Page table (PT) holds num of entries.
* Num of entries = num of pages in LAS
* Page table entry contains frame number in PAS.
* Page table size = $N * e$; e → entry space in Byte ($e \geq 1B$)
* Page table is stored in main memory.
* Each process has its own PT, it contains the base address of the frame location for each page in the process.

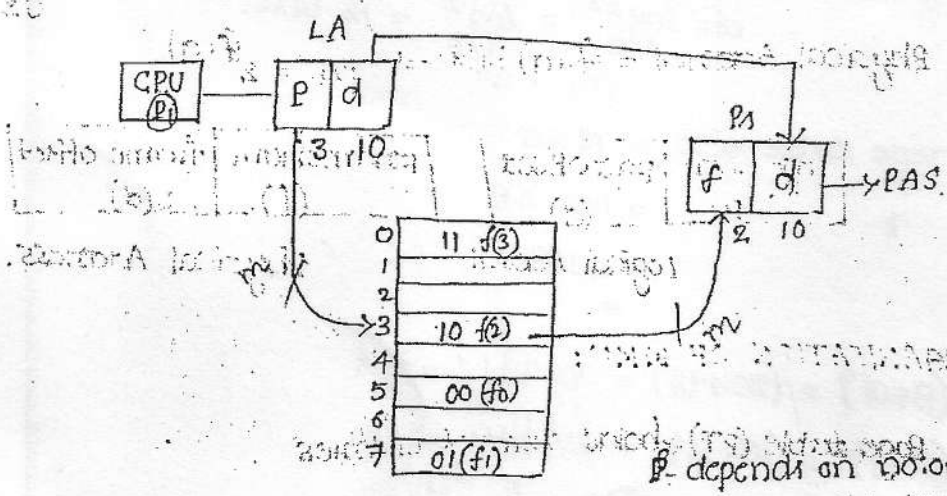


* Page table base reg (PTBR) points to page table.
* page table length reg (PTLR) indicates the size of table.
* in this schema data/instruction access requires 2 level memory access (PT+MM).

∴ Time consuming ⇒ solution ⇒ TLB

(Translation Lookaside Buffer)

Address Translation:



⇒ p depends on no.of pages

d - offset.

eg:

(9) if 1000 KB memory is managed by using variable partitions but no compaction it currently has 2 partitions of size 200 KB & 260 KB

The smallest allocation request in KB that could be denied for.

A. 151

(B) 181

C. 231

D. 541

200 + 260 = 460

1000 - 460 = 540

540/3 = 180

LA = 32 bit
PA = 27 bit
PS = 4KB
c = 4B

LA = 32 bits

$LAS = 2^{LA}$

$= 2^{32}$

= 4GB

LAS = 4GB

PAS = 128MB

PT entries =

PA = 27 bits

$PAS = 2^{PA}$

$= 2^{27}$

= 128MB

↳ No. of pages: $N = \frac{LAS}{PS} = \frac{2^{LA}}{2^{PT}} = \frac{2^{32}}{2^{12}} = 2^{20}$ = 1M pages

↳ No. of frames: $M = \frac{PAS}{PS} = \frac{2^{PA}}{2^{PT}} = \frac{2^{27}}{2^{12}} = 2^{15}$ = 32K frames.

$p = \log_2^N = \log_2^{2^{20}} = 20$ bits

$f = \log_2^M = \log_2^{2^{15}} = 15$ bits

$d = \log_2^{PS} = \log_2^{2^{12}} = 12$ bits.

↳ page table size = N * e

= 1M * 4B

= 4MB.

=> LAS = (p+d) = (20+12) = 32 bits = LA

=> PAS = (f+d) = (15+12) = 27 bits = PA.

PS = 2K
No. of pages in N
No. of frames = M
PAS = 4MB
e = 4B

N =
M =
PA =
d =
p =
f =
LA =

$p = \log_2^N = \log_2^{2^{11}} = 11$

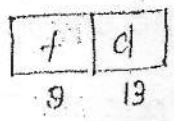
$f = \log_2^M = \log_2^{2^9} = 9$

PAS = 4 MB

PA = 22 bits $\Rightarrow$ f = 9 $\Rightarrow$ f + d = 22

d = 13

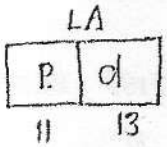
d = 13



LAS = P + d

= 11 + 13

LAS = 24 bits.



Page table size = N * E

= 2K * 4B

= 8KB.

Eg: consider a computer sys has 2K pages & 256 frames calculate the size of ~~LAS~~ p & d if ~~LAS~~ = PAS = 1MB.

$\Rightarrow$ N = 2K pages

M = 256 frames

$PA = \log_2^{PAS}$

$= \log_2^{2^{20}}$

= 20

$P = \log_2^N$

$= \log_2^{2^{11}}$

P = 11

$f = \log_2^M$

$= \log_2^{2^8}$

= 8

~~LA = P + d~~

~~20 = 11 + d~~

~~d = 9~~

PA = f + d

20 = 8 + d

d = 12

Eg: consider a system with 32-bit logical address and 64 MB physical address space, page size is 4KB. calculate approximate page table size in bytes. and E = 2B

LA = 32 bit PS = 4KB

PAS = 64 MB $= 2^{12}$

$PA = \log_2^{PAS} = \log_2^{2^{26}} = 26$ bits

$N = \frac{LAS}{PS} = \frac{2^{LA}}{PS} = 2^{32}/2^{12} = 2^{20} = 1M$

Page table size

= E * N

= 2B * 1M

= 2MB.

Translation Lookaside Buffer (TLB):

* A special, small, fast lookup hardware cache (Associative cache)
* It holds some previously/recently used page table entries while the actual page table continue to reside on main memory.

* The page is numbered first searched for in TLB

  if found → TLB => HIT

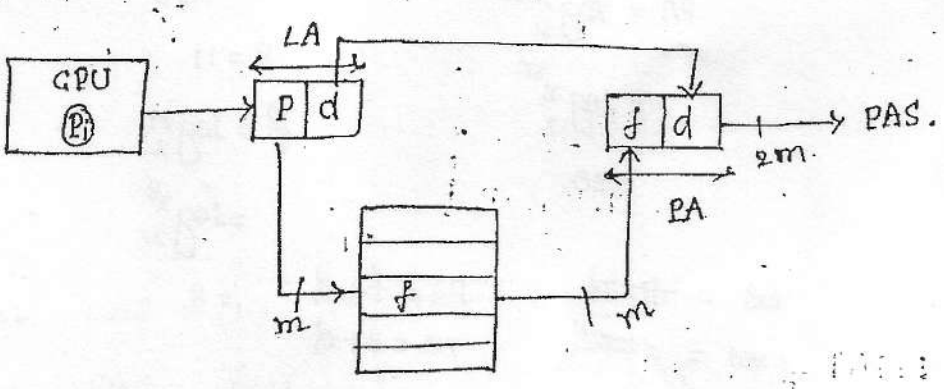
  otherwise → TLB => MISS.

Performance of paging:

Main memory access time (MMAT) = m (usec/nsec)

Effective memory access time = 2m

* To decrease/reduce of EMAT from 2m to m to bring cache
* Access time is amount of time to read or write to memory.



Page table.

Paging with Translation Lookaside Buffer:

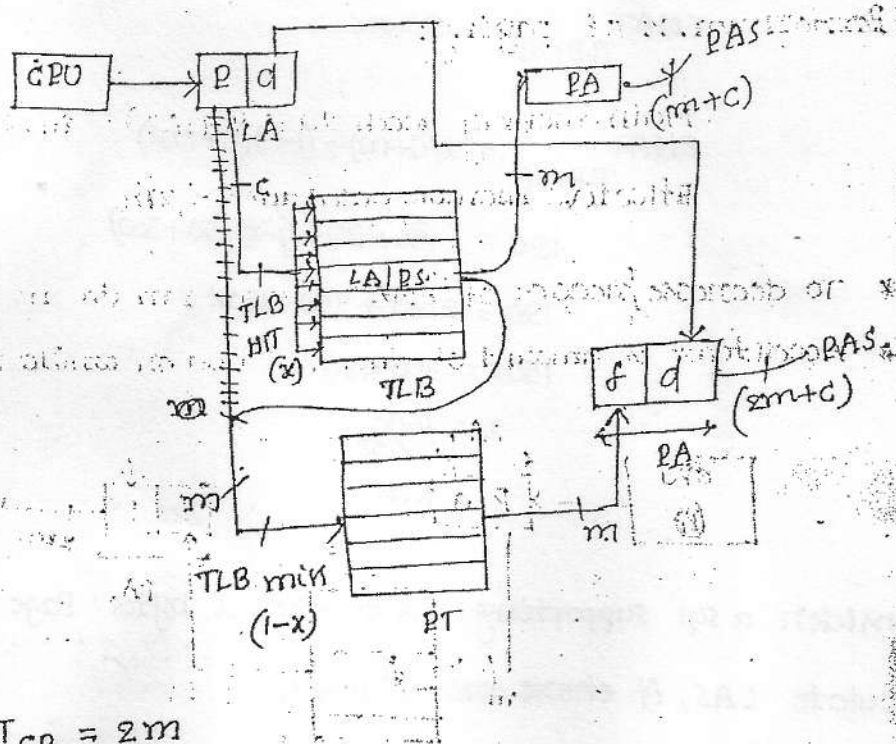
* A TLB access time is less compared to MM. The over all access time to get the page from physical memory.
* TLB is a cache contains list of physical and logical address. if generated LA is present then it is TLB hit (m) it goes to page table read

* if generated LA is not found within the acache, or it is called "TLB miss" if it found is called "TLB HIT".

TLB acces time = c (c<<m)

TLB page hit ratio = x = No. of hits / Total reference

$$EMAT_{TLB+SP} = \underbrace{x(c+m)}_{\text{succes}} + \underbrace{(1-x)(c+2m)}_{\text{failure}}$$



NOTE: $EMAT_{SP} = 2m$

$EMAT_{SP+TLB} = x(c+m) + (1-x)(c+2m)$.

* Both PT and PAS is in physical memory. Thus, a TLB miss results in 2 acces in physical memory.

Eg: TLB hit ratio = 80% = 0.8

TLB acces time = c = 20 nsec

Page table acces time = m = 100 nsec

EAT = 0.8 (20+100) + (1-0.2)(20+2(200)) nsec

= 140 nsec.

* in hit ratio translation requires "c+m" if main access it takes "c+m+m = "c+2m"

Eg: what hit ratio is required to reduce the EMAT from 300 nsec (without TLB) to 190 nsec (with TLB). Assume cache access time is 50 nsec

↳ c = 50 nsec

x =

2m = 300 ⇒ m = 150

$EMAT_{TLB+SP}$ = 190 nsec

$EMAT_{TLB+SP} = x(c+m) + (1-x)(c+2m)$

$190 = x(50+150) + (1-x)(50+300)$

$190 = 200x + 350 - 350x$

$150x = 350 - 190$

$x = 160/150$

$= 1.06$

Eg: consider a sys supporting PA of size 'x' bytes Page size is 2^K B calculate LAS, if there are 2^P pages.

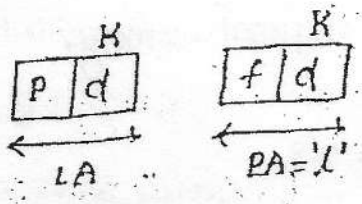
PA = 'x' bytes

PS = 2^K

LAS = 2^{P+K}.

$d = \log_2^{PS} \Rightarrow d = \log_2^{2^K}$

$d = K$



LA = P+K

LAS = 2^{P+K}

MULTILEVEL - PAGING / Hierarchical paging:

As the memory address length increases, the LAS also increases, making the page table size becomes larger and larger.

Eg: With 32-bit OS the LAS = 2^{32}.

If we take the pagesize ~~table~~ to be 2^{12} B = 4KB

Num. of entries in page table = $\frac{2^{32}}{2^{12}} = 2^{20} = 1MB$

Now one frame number being 32-bit = 4B long, the

Page table size = 1M * 4B
= 4MB

* Large page table consumes more main memory

∴ Therefore we can use Multilevel paging.

* The page table itself is paged again, creating a outer Page table and an inner page table.

Reduce PT size:

→ Reducing the no. of pages (N) by increasing pagesize.

eg: Pagesize = 1026 B

2 B	1024 KB
small	large
↓	↓
513 pages	2 pages
IF = 0 Bytes	IF = 1022 B ≡ 1 Page.

→ Multilevel paging: Main objective is reduce the PT size

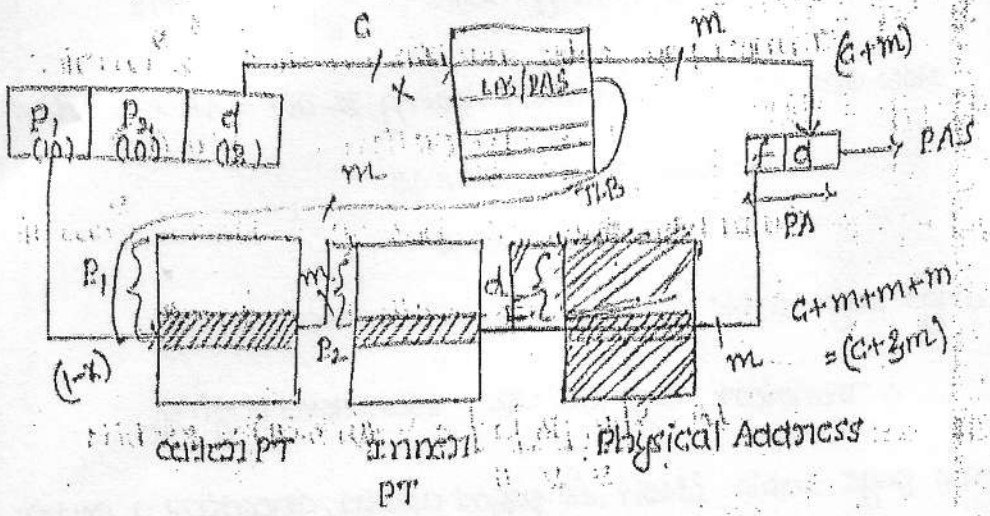
→ MLP applies paging on PT

* Dividing the PT address into equal page sizes.
* Store the pages of PT in MM frame.

→ Access the pages of PT in PAS through another PT.

* The new PT is known as outer level PT and the original one is known as second level page table.

In the above example, the 30-bit of single page offset can be broken/divided into 3, 10-bit long page offsets, one for inner Page table, other for outer page table



* First, locate the inner page table's page reference in the outpage table using (P_1 bits).
* Next, in the inner page table, locate the frame reference, using (P_2 bits).
* Forward mapped page table.
* Finally locate the in the physical address using "d" bits.

Performance: MMAT = m

$EMAT_{1LP} = 2m$

$EMAT_{2LP} = 3m$

$EMAT_{NLP} = (n+1)m$,

TLBAT = c

TLB HIT = x

TLB MISS = (1-x)

$EMAT_{TLB+2LP} = x(c+m) + (1-x)(c+3m)$

$EMAT_{TLB+nLP} = x(c+m) + (1-x)(c+(n+1)m)$.

Eg: consider a paging schema where pagesize is 2KB, outer & inner page table is divided into 512 pages each of size 4B. calculate (i) size of LAS/VAS and PAS

(ii). offset of inner page table size.

∴ page size = $2KB = 2^{11}B$

$d = 11$ bits

Inner page table has 512 entries = 2^9 entries.

inner page offset = 9-bits (P_2)

outer page table also has 512 entries = 2^9 entries.

outer page offset = 9-bits (P_1).

LA → | P_1 (9) | P_2 (9) | d (11) | ⇒ (9+9+11) = 29 bits

$LAS = 2^{LA} = 2^{29} = 512MB$.

Each entry of inner & outer page table has 4B.

outer page table size = $2^9 \times 4B = 2^{11}B = 2KB$

inner page table size = $2^9 \times 4B = 2^{11}B = 2KB$

PA → | f (4) | d (11) | ⇒ (4+11) = 15-bits.

$PAS = 2^{PA} = 2^{15} = 32KB$.

Total size of all inner page tables = $(2^{9+9}) \times 4B$

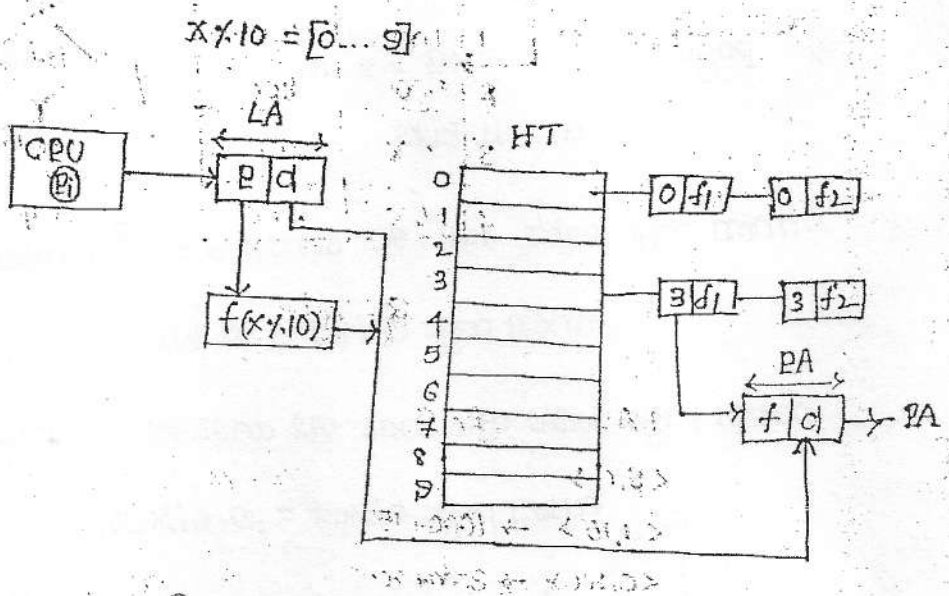
= 1MB

(2^{9+9} ← No of entries; 4B ← length of each entry)

PAGING WITH HASHING:

→ Process is not associated with any PT, but it is associate with a Hash table.

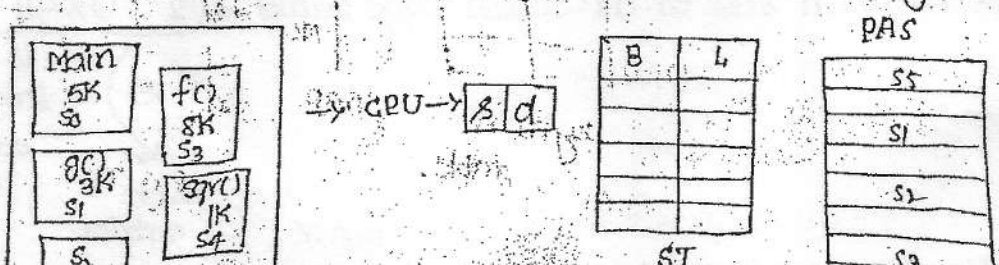
→ collision resolution with chaining.

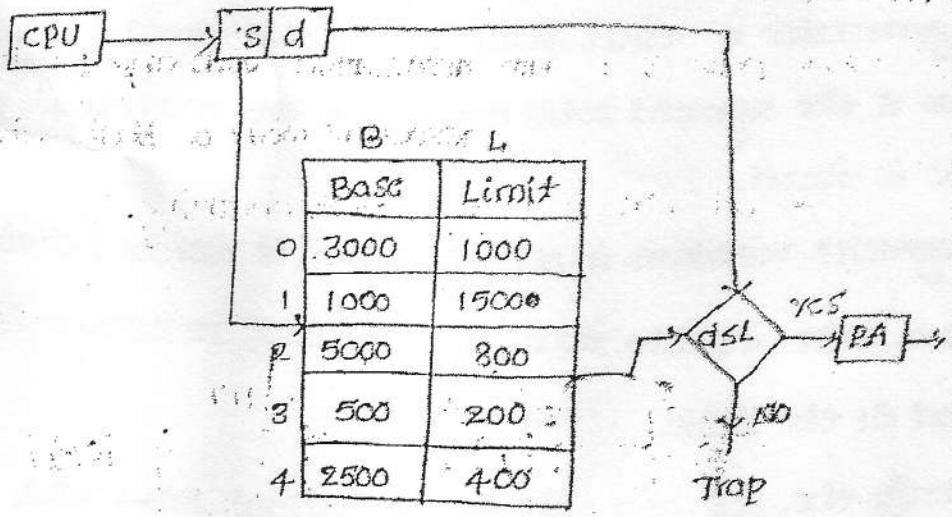


DisAdv: PAGING Does not preserve users view of Mem. Alloc.

SEGMENTATION:

* Segment is a piece of code or module
* The program is divided into several segments (ie seg main, subroutine, symbol table, library functions, stack)
* The process memory is also divided into corresponing segments, which in turn, is furthen divided into pages.
* It preserves the user view of memory.
* Each segment has name and length.
* Logical address space is nothing but a collection of segment.





Seg Table

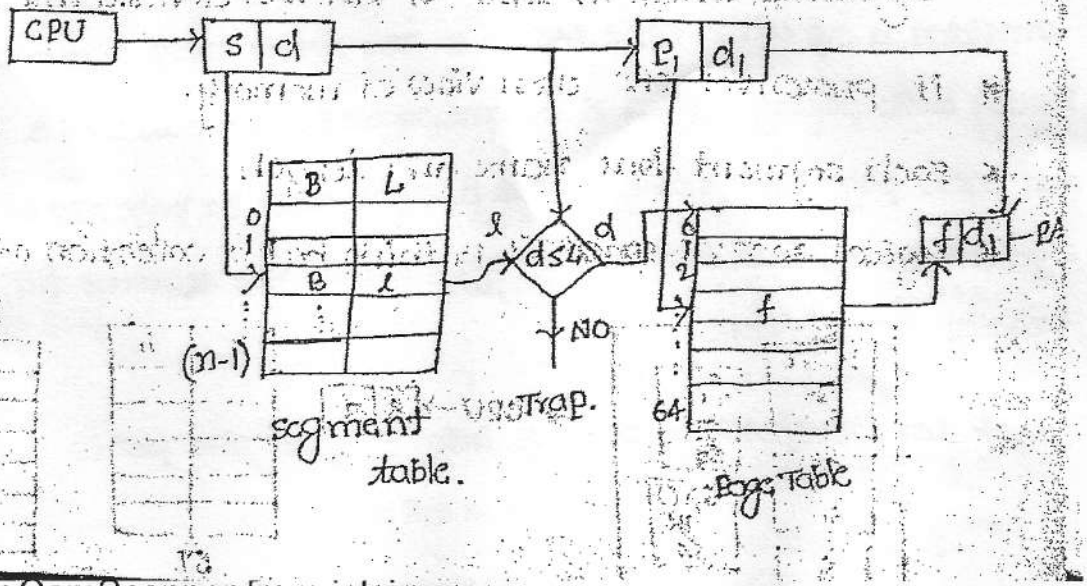
LAS	PAS
<s,d>	
<1,15>	→ 1000+15
<0,200>	→ 3000+200
<4,4000>	→ Trap
<2,950>	→ Trap
<1,1000>	→ 1000+1000.

=

Segmented paging:

* This is segmentation along with paging. Logical Address is divided into 2 parts, segment offset(s) + page offset(d)

word offset(d)



Maximum segment size $= 2^8$

$= 256.$

$LAS = 64KB.$

If LAS is divided into 8 equal sized segments then

then max segment size $= 2^{16}/8$

$= 2^{13} B$

$= 8KB.$

↳ Number bits in page offset being K, say page size becomes 2^K

Num. of page : $N = \frac{LAS}{2^K} = \frac{2^{13}}{2^K} = 2^{13-K}$

∴ page table size $= N * e$

$= 2^{13-K} * 2$

$= 2^{14-K}$

By Problem page table size = page size

$2^{14-K} = 2^K$

$14-K = K$

$K = 7$

Page table size $= 2^{14-K} = 2^{14-7} = 2^7 = 128 B.$

Optimize Tech;

=

Eg: LAS/VAS = 's' words

Page table entry = 'e' bytes

Page size = P

PT overhead (Byte) $= \left(\frac{S}{P}\right) * e$ (N * e, size)

internal frag $= (P/2)$ [on average]

Total overhead $= \left(\frac{P}{2} + \frac{S}{P} * e\right)$

To minimize/optimize $= \frac{dl}{dp} = \left(\frac{P}{2} + \frac{S}{P} * e\right)$

$0 = 1/2 - 1/P^2 S * e$

$P = \sqrt{2Se}$

Paging Vs Segmentation:

	Internal fragm	External fragm
Paging	YES	NO
Segmen	NO	YES

VIRTUAL MEMORY.

In general: VAS/LAS > PAS & secondary storage > VAS.

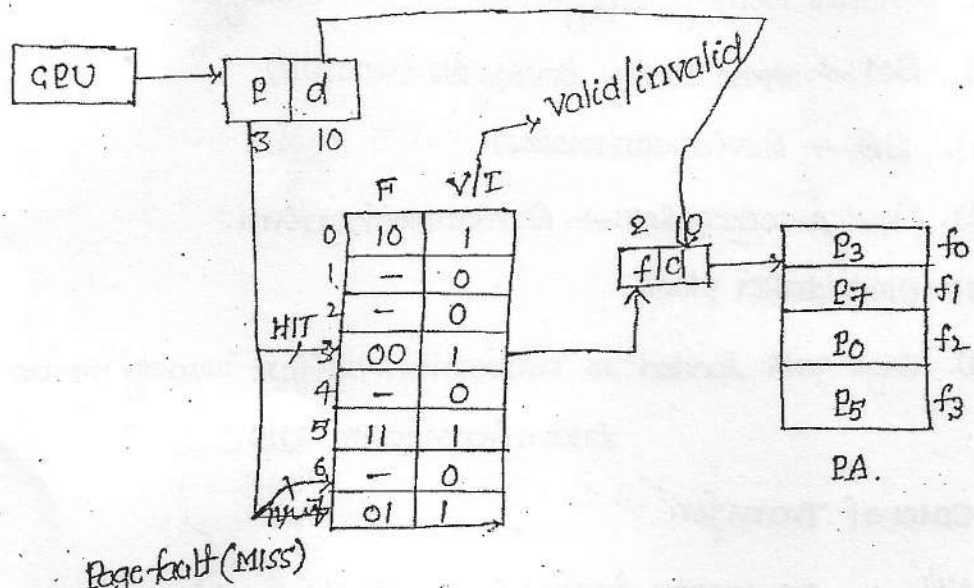
i.e PAS < LAS/VAS < secondary storage.

∴ Main memory < virtual memory < secondary storage.

LAS/VAS = 8KB

PAS = 4KB

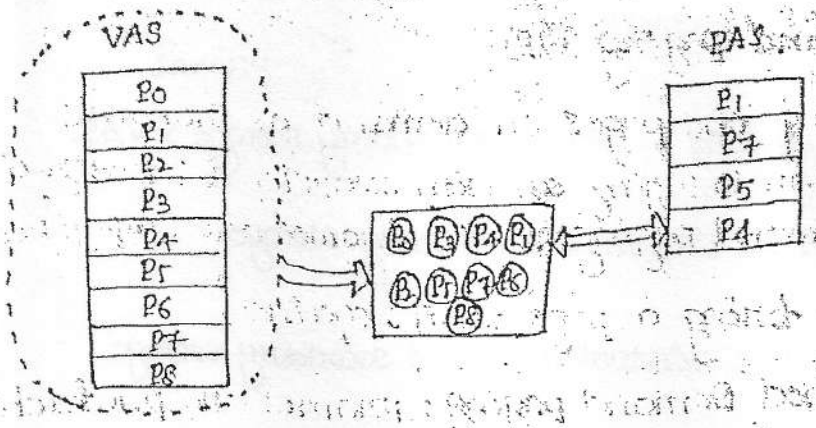
PS = 1KB



** virtual memory gives an illusion to the programmer that a huge amount of memory is available for writing a program in which is largen than main memory.

* The execution of the process, the pages of largen program are controlled dynamically by OS with the help of DMA.

* The valid/invalid bit associated with every frame refarm tells the OS. that a frame requested by CPU is actually residing in MM or not, if not a page fault has occured



* Page fault is handled by "Page Fault Service". It follows the following sequence.

(i). Process causing page fault get blocked.

(ii). Virtual Memory Manager → Device Manager

(iii) ~~VMM~~ DM → Device Driver (DD)

(iv). DD → Device controller

(v). Device controller → Device Mechanism.

(vi) DMA takes place.

(vii). Page gets loaded to controller buffer ready to be transferred to MM.

Case of Transfer:

(viii) → An empty frame is available (copy)

→ No empty frame

↳ Page replacement.

(ix). Page table is updated

(x). Unblock the process.

* If the page to be moved out (victim page) is "clean" i.e it has not been modified while in MM. no problem PUSH it out.

* Otherwise, the page has been modified ("dirty"). Before Pushing out. its latest form has to be written back to SS.

Demand paging (DP)

Loading the pages on demand at runtime

→ similar to paging sys with swapping & Process resides

=> Pure Demand paging: All the frames are empty (Don't bring a page until required)

=> prefetched Demand paging: Frames are prefetched (Bring some pages based on prediction)

Performance:

(i) Effective Memory Access Time (EMAT)

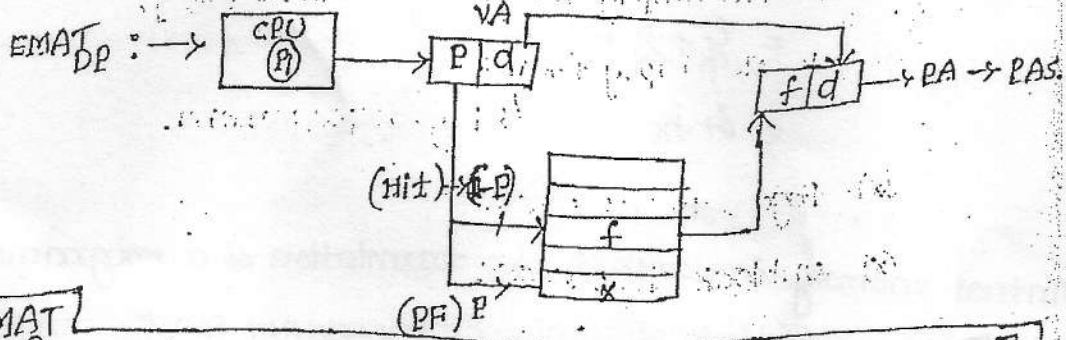
→ Main Memory Access Time = "m" nsec

→ Page Fault Service Time = "s" msec (m<<s)

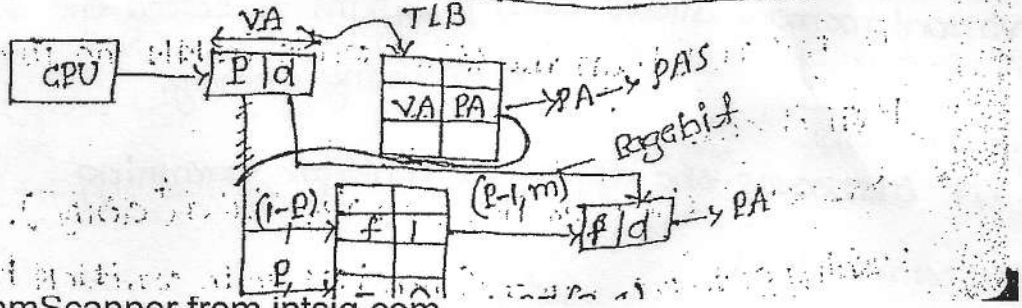
→ Page Fault Ration: p = No. of Page Faults / Total reference

→ Page Hit Ratio: (1-p)

$$\therefore EMAT_{DP} : p*s + (1-p)m$$



EMAT Demand paging + TLB = $x(c+m) + (1-x)[(1-p)m + (p*s)]$



g: Suppose the time to service a page fault on the avg is 10 msec while a memory access takes 1 μsec. Then a 99.99% hit ratio results in avg mem access time of.

↳ MMAT = m = 1 μsec

PFST = S = 10 msec

Page fault hit ratio : (1-P) = 0.9999

Page fault ratio : P = 0.0001

$$EAT = P*S + (1-P)m$$
$$= 10\text{ msec} * 0.0001 + 0.9999 * 1\ \mu\text{sec}$$
$$= 1.9999\ \mu\text{sec}.$$

Eg if an instruction takes 'i' msec and page fault takes an addition 'j' msec, The effective instruction time if on avg a page fault occurs every 'K' instruction is:

↳ m = i msec

S = (i+j) msec

P = 1/K , 1-P = 1 - 1/K

$$EAT = \frac{1}{K}(i+j) + \left(1-\frac{1}{K}\right) i$$
$$= \frac{i}{K} + \frac{j}{K} + i - \frac{i}{K}$$
$$= i + j/K$$

19.

NOTE: virtual memory implements the translation of a programs address space into physical memory address space.

* virtual memory allows each program to exceed the size of the primary memory.

* VM increases the degree of multiprogramming.

PAGE REPLACEMENT TECH:

(A). page Reference string: set of successively unique pages referenced in the given list of logical address.

Logical Address:

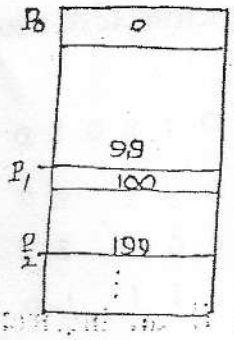
463, 182, 134, 195, 435, 963, 967, 834, 128, 534, 765, 784, 012, 624, 68

Page size = 100 w consider

463 => page address

$\frac{463}{100} = 4$ (every Page contain 100 words)

463 % 100 = 63 -> offset



= {4,1,1,1,4,9,9,8,1,5,7,7,0,6,6} => if occurred successively then count only once.

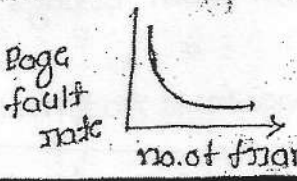
Reference string = {4,1,4,9,8,1,5,7,0,6} => 4 occurs again but not successively.

n = length = 10

↳ Num. of unique pages referenced in the given list of logical addresses:

n = {0,1,4,5,6,7,8,9} = 8

choosing a victim page for replacement is a critical and final decision. it needs to consider factor like how frequently the page is used and whether the page has been modified (Dirty) or not. to reduce page faults as per as possible.



No. of frames ∝ page fault rate.

FIFO :

Based on Arrival time.

* FIFO page replacement algorithm selects the page that has been memory the longest.

Reference string:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

frame size: 3

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	4	4	4	0		0	0		7	7	7			
	0	0	0	3	3	3	2	2	2		1	1		1	0	0			
		1	1	1	0	0	0	3	3		3	2		2	2	1			

∴ page faults : 15

if frame size : 4

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	7		3		3				3			2		2			
	0	0	0		0		4				4			4		7			
		1	1		1		1				0			0		0			
			2		2		2				2			1		1			

Page faults : 10

Belady's Anomaly: with increase of page frame to a process the page fault rate also increases sometimes it is called "Belady's Anomaly".

* FIFO based policies suffer from "Belady's Anomaly"

↳ Frame increase => page fault rate increase => Anomaly

↳ Belady's Anomaly => unusual behaviour.

optimal page Replacement:

* An algorithm that keeps track and predict how frequently a page is going to be used and hence, replace those pages which are not going to be used frequently.
* lowest page faults of all algorithms, But it is not implemented.

" Replace the page will not be used for longest Period of time."

Reference string:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Frame : 3

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	2	2	2	2	3	3	3	2	3	3	3	2	1	1	1	1	7	7	7
0	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
				0		0				0					0			0	

Page faults = 9

Frame : 4

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	3	3	3	2	2	2	3	3	3	2	2	2	7	7	7
1	1	1	1	1	1	1	4	4	4	0	0	0	0	0	0	0	0	0	0

Page faults = 8.

* optimal page replacement does NOT suffer from the "Belady's Anomaly."

Least Recently Used:

* Partial implementation of optimal algorithm.
* LRU Policy replace the page in memory that has not been referenced for the longest time.

Reference string : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

frame : 3

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	3	3	3	2	2	2	3	3	3	2	2	2	7	7	7
1	1	1	1	1	1	1	4	4	4	0	0	0	0	0	0	0	0	0	0

Page faults : 12

Most Recently Used (MRU)

counting / stack Algorithm:

→ Least Recently used (LRU)

→ Most Recently used (MRU).

* A reference counter value is assigned to every frame denoting the no of time a frame is referenced.
* Replace the page having least/most number of references.
* Free from "Belady's Anomaly"
* As uses, LRU (or) LRU approximation.

LRU Approximation:

→ A variation of LRU

→ Every page table contains a reference bit within it.

→ A Reference bit is associated with each entry in the page table It is set by the HW whenever the page is referenced. (Read/write).

→ Replace the page which has not been referred at all. (R-bit = 0)

→ if need to replace a page with R bit = 1, then set it to '0' again after moving in the new page.

→ it fails when all the R-bits of page table entries are 1.

Page Table

P	O	V/I	AT	R
0	a	1	3	1
1	b	1	2	0
2	-	0	-	-
3	d	1	0	1
4	K	1	1	0

R < 0 → page has not been referred so for during the present epoch
1 → Page has been referred atleast once during the present epoch.

→ At the end of epoch reference bits are cleared to "ZERO"

EPOCH => Duration of time => time Quantam

⇓

Time divided into different discrete intervals & called "epoch".

→ During current epoch, what pages are referred (or) not referred are indicated by reference bit(R).

	R
P_1	0
P_2	0
P_3	0

=> initially

	R
P_i	1
P_j	0
P_k	1

→ page referred in new epoch.

⇓

End of old epoch & starting new epoch has initial value = 0.

* LRU approximation is "Reference bit algorithm".

* if all reference bit value = 1, then this algor fails

⇓

Second change Algor.

Second chance/clock Algorithm:

criteria : Arrival time + Reference bit

(AT + R)

* FIFO based algorithm.
* Implemented as " circular Queue".
* Starts the search of pages from AT and if it is already referred then give a second chance to it.
* value changed from 1 to 0.
* When searched for replacement, replace, a page which has R- bit = 0, if any.
* otherwise, look for a page considering arrival time (FIFO). if any page is found with R-bit = 1, it is given "a a "second chance" and search continues.
* suffers from "Belady's Anomaly"

P	F	V/I	AT	R	
0	a	1	3	✗ 0	+④
1	b	1	2	✗ 0	-③
2	-	0	-		
3	d	1	0	✗ 0	-①
4	k	1	1	✗ 0	-②

Enhanced Second chance: Avoids unnecessary page both (Not recently used)

criteria : Referred bit + Modified bit

(R+M) ⇒ Dirty bit.

Modified bit => checks whether the contents of page are modified or not.

M = 0 => Page is clean, No write has been done

M = 1 => Page has been written/modified => Become "Dirty"

R M

0 0 → Neither reference not modified.

0 1 → Not referred but modified

1 0 → Referred but not modified.

1 1 → Referred & Modified.

R=0, M=1:-

	R	M	R	M
P_i →	✗	1	0	1

0

↳ it is referred in old epoch and it is not referred in new epoch but modified.

* when page fault occurs consider the value "00", we look for '01' combination next, since it represent LRU, then '10' is considered & at last "11" is considered.

Page Allocation Algorithm:

Proportional Allocation:

* if there are n process P_i $(0 \le i \le n-1)$ with m free frames and size of each process is S_i then allocate the num. of frames to each P_i which is proportional to S_i.

* Total size of all process $= S = \sum_{i=0}^{n-1} S_i$

* Process P_i is allocated num. of frames $= \left(\frac{S_i}{S} * m\right) = n_i$ by rounding the n_i to the nearest whole number.

Eg: m=50, n=5

$$\sum_{i=0}^{4} S_i = (10+3+15+30+17) = 75$$

P_i	S_i	Prop. Alloc	
P_0	10	6	$n_0 = \frac{10}{75} \times 50 = 6.66$
P_1	3	1	$n_1 = \frac{3}{75} * 50 = 1.33$
P_2	15	10	$n_2 = \frac{15}{75} * 50 = 10$
P_3	30	20	$n_3 = \frac{30}{75} * 50 = 20$
P_4	17	11	$n_4 = \frac{17}{75} * 50 = 11.33$

50% Rule:

* In this method OS checks the no. of unique referenced by the process.
* Allocates 50% of the total no. of unique pages.

Eg: Page reference string: {4,8,4,5,3,4,9,0,1}

No. of unique pages = n = 7 => {0,1,3,4,5,8,9}

Allocated no. of frames/Pages = (7/2) = 3.

Mapping of logical Address into page:

A simple technique by which the logical address would be mapped into the page.

$$\text{Page number} = \left(\frac{\text{logical Address space}}{\text{Page size}}\right) = P$$

$$\text{Page offset} = (\text{Logical Address} \% \text{ Page size}) = d$$

Eg: LA = 443 => P = (443/100) = 4

PS = 100

d = (443%100) = 43.

Q: Find the page location for the following LAs issued by CPU.

{443, 864, 864, 872, 438, 512, 518, 338, 394, 401, 936, 012, 112}

P = {4, 8, 8, 8, 4, 5, 5, 3, 3, 4, 9, 0, 1}

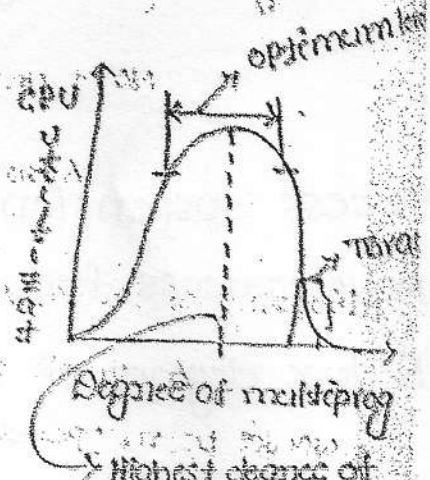
d = {43, 64, 64, 72, 38, 12, 18, 38, 94, 01, 36, 12, 12}

Num. of unique frame = n = 7 => {0,1,3,4,5,8,9}

=> By 50% algo, we need to allocate only = (7/2) = 3 frames.

THRASHING:

* Excessively high rate of paging activity and high rate of page faults.
* Excessive/ high page fault rate is called "thrashing".
* Thrashing is also undesirable state of the sys like deadlock
* CPU utilization is very poor,
* System performance is poor.
* It exists in a feedback mech in multiprogramming environment.



Reasons for thrashing:

Primary Reasons:

* lack of frames (memory)
* High degree of multiprogramming
* Inefficient OS.

Secondary Reasons:

* selection of page size → small → more pages → more faults
  → large → less pages → less faults.
* page replacement algm.
* program structure / Data structure / Prog Tech.

Thrashing Control Strategies:

Type 1: Prevention:

→ control the degree of multiprogramming.

Type 2: Detection & Recovery:

Detection: * CPU utilization is poor.

gh degree of multiprogramming. control
h utilization of paging device / page disk / swap partition

very:

→ Process suspension

→ Move process from MM to secondary storage.

→ Reduce degree of multiprogramming.

mporal (Time) → Thrashing.

atial (space) → internal fragmentation.

entrant code can be shared by all programs and non-self modifying code.

ging is a method of memory allocation by which the gram is subdivided into equal portions and core is divided into equal portions of blocks.

rashing can be decreased by decreasing the degree of multiprogramming.

FILE SYSTEM & DEVICE MANAGEMENT

* Interface :

CPU — Purely electronic

HDD — Electro Mechanical

* Every secondary / Tertiary media must have their own file sys.

→ we need a controller interface as controller card / chip ⇓ H/w requirement

controller card

IDE — Intelligent Device electronics ⇓ Desktop

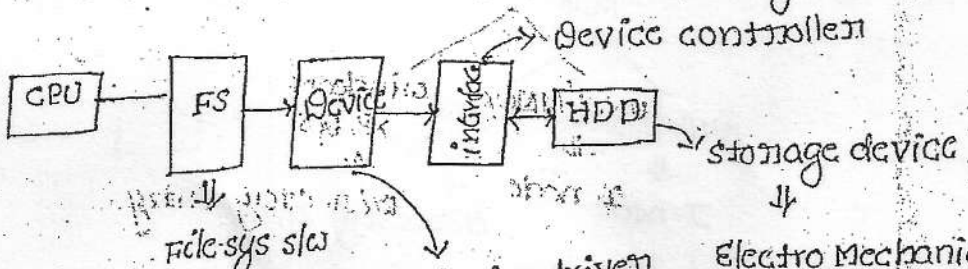
SCSI — small computer system interface ⇓ Server.

SATA

ATA

→ Software Requirement ⇒ Device driver

→ Device independent software ⇒ called as "file system".



File: collection of logically related set of records of an entity
file is a data structure.

Representation / implementation:

* Flat service of Bytes ⇒ UNIX, LINUX
* Record structure ⇒ Trees, B-Tree, B+ Tree.
* Tree structure.

Operations:
- create
- open
- Read
- write
- Modify
- close
- Delete
- Repositioning
- Truncation
- Renaming.

Attributes:
- File name
- Identifier
- Type, size
- path, Mode
- protection
- Date, Time
- User ID.

⇓ stored in FCB

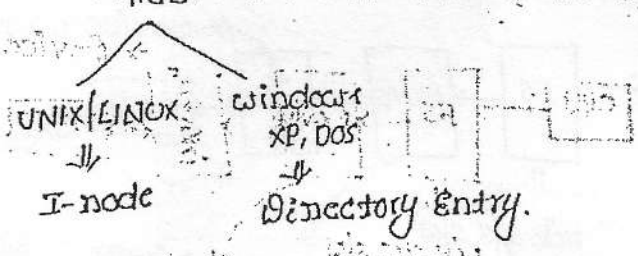
Types:
- Text
- object
- program
- Video
- Audio
- Compressed
- Multimedia
- Data
- Library
- executable.

Access Methods:

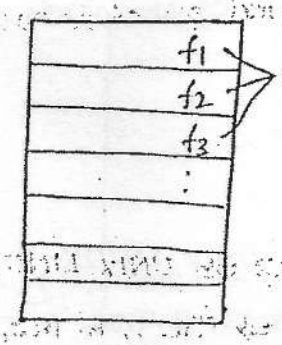
Sequential Access ⇓ Slow (TAPE)

Direct / Random ⇓ FAST {A set of fixed length logical records}.

* Attributes of a file are stored in FCB



Directory: contains the attributes of a file/ information of a the file, ie Metadata of the file.



* Directory acts as a symbol table that translates file name into their directory entry.

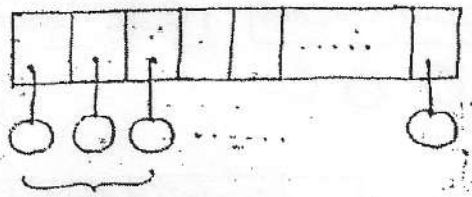
supported operations:

- → search for a file
- → create a file
- → list a directory
- → Rename a file
- → Traverse the file system.

Directory structure:

* Single level directory
* Two level directory.
* Multilevel / tree structure
* Acyclic Graph.

Single level Directory:



Adv: → simplicity
→ searching
→ no Name conflicts

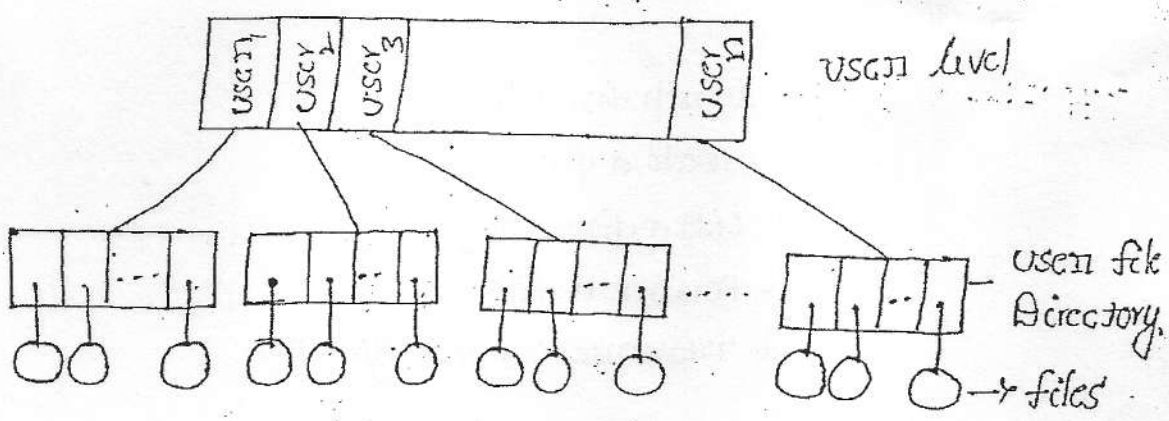
Dis Adv: N-entries - N-files => not more than N

Searching is sequential, so as the no. of entries increases, search time/access time also increa

→ No 2 files can have same name.

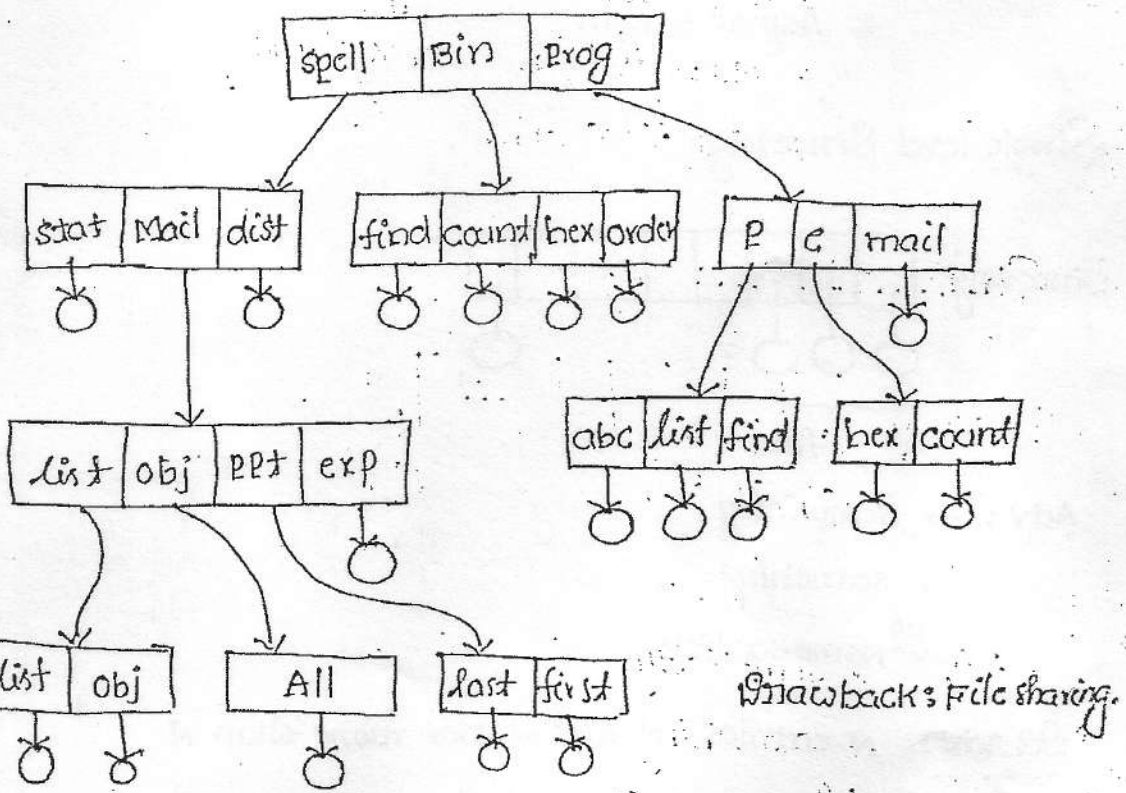
→ Two different users, simultaneously can have same file name.

Two-Level Directory:



creating different files with same name → flexibility

Multilevel Directory / Hierarchical / Tree Structure Directory:

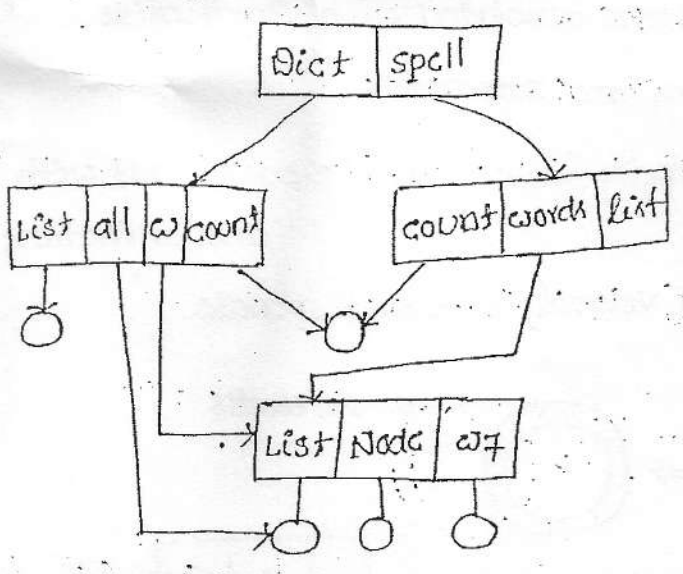


Drawback: File sharing.

* sharing by duplications has drawbacks of → inconsistency
→ wastage of space.

* To overcome the duplication of file sharing, we have the concept of file sharing with links called "Directed Acyclic Graph".

Directed Acyclic Graph:



DEVICE CHARACTERISTICS:

Schematic Disk Assembly:

* More flexible, but more complex
* A link pointer could be use to make a file shariable among users.
* Tree structure does not permit sharing of a file. Acyclic graph permits that. Thus we need this type of directory structure to implement shariable file.

* Major component in secondary is Disk:

Physical structure/ Geometry of Hard Disk:

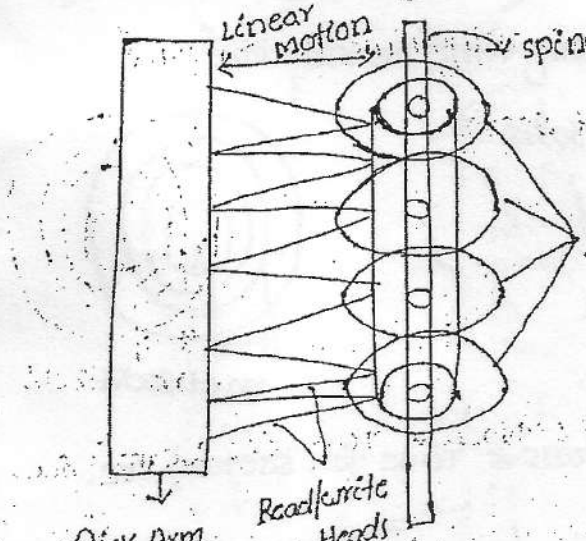


fig: Disk structure.

→ Hard disk consists of set of platters with a spindle.

→ Each surface of platters associated with Read/write head.

→ two types of motions are supported

* Linear velocity (move forward & backward) with disk armature

* Angular velocity with disk spindle.

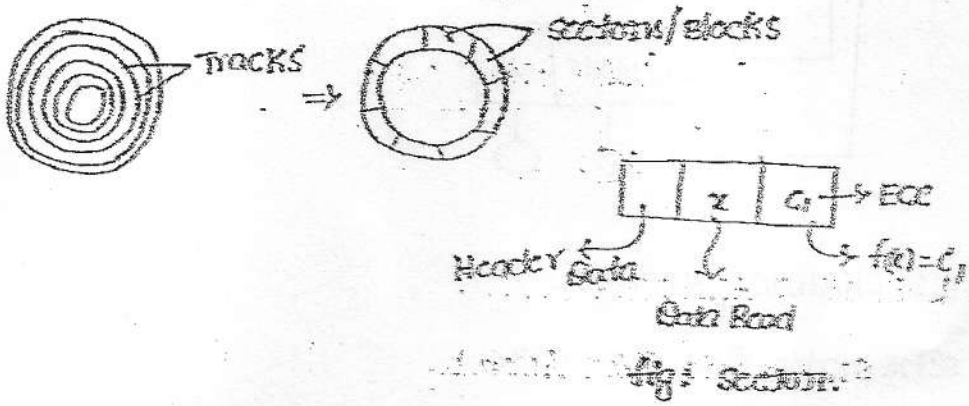
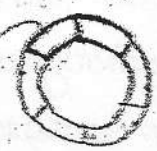


fig: Section.

Cluster: Group of one or more adjacent sectors. It is the unit of transfer from disk to MM.



Cylinder: Num of tracks on the surface. All the same positioned tracks of all the platters together forms a logical cylinder.

Seek time (ST): Amount of time needed by the Read/write head to move the disk arm assembly to the desired cylinder.

* It is also called "Positioning time/ Random time"

* Seek time of no. of sectors

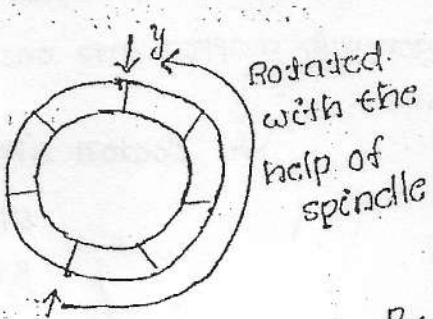
ST = (No. of sectors * time for sector) sec.

=

Latency Time (LT): Amount of time taken to rotate a section in order to position it below the disk head.

* it is also called "Rotational Delay Time."
* it is calculated as average time of rotation i.e. $R/2$

R = Rotation time.



Eg: Disk RPM = 3600

$$R = \frac{1}{3600} \times 60 \text{ sec}$$

$$= \frac{1}{60} \text{ sec}$$

$$\Rightarrow LT = \frac{R}{2} = \frac{1}{120} \text{ sec}$$

Transfer Time: (TT): Transfer rate is the rate at which data flows from disk drive to another components (like MM).

Transfer time depends on → Disk size
→ Section size
→ Rotation Time.

suppose track size = S Bytes

Section size = B Bytes Section size < Track size

Rotation time = R sec.

⇒ in R sec disk can read on Track i.e "S" Bytes.

Time taken to read one section i.e "B" Bytes

$$= \left(B * \frac{R}{S}\right) \text{ sec}$$

$$\therefore TT = \frac{BR}{S} \text{ sec.}$$

Disk I/O Time: (Seek time) + (Latency time) + (Transfer time)

$$= (ST) + (LT) + (TT) \text{ sec}$$

$$=$$

Q: How long does it take to load a 64-KB prog. from a disk whose avg seek time is 30 msec, Rotation time is 20 msec, track size is 32 KB & page size is 2KB? Assume that the pages of the prog are spread randomly around the disk. page size is generally mapped into one sector size.

↳ Section size = 2KB = B.

ST = 30 msec

RT = 20 msec

Program size = 64 KB.

⇒ No. of Pages = $\frac{64\text{KB}}{2\text{KB}}$ = 32 = num. of sections to be transferred.

$\frac{LR}{2} = \frac{R}{2} = \frac{20}{2} = 10$ msec.

Track size = 32 KB. = S

$TT = \frac{BB}{S}$

$= \left(\frac{20 * 2}{32}\right)$

$= 5/4 = 1.25$ msec.

Time needed to transfer 1 section through Disk I/O

$= ST + ET + TT$

$= (30 + 10 + 1.25)$

$= 41.25$ msec.

Time to transfer 32 sections = (41.25 * 32) msec

= 1320 msec

= 1.32 sec.

Eg: consider a Disk drive with the following specification:

16 platters

↳ 2 surface

↳ 512 Tracks

↳ 2K Sector

↳ 4KB page/sector size.

(i) capacity in Byte =

= Num of sectors in disk drive * Section size

= $(16 * 2 * 512 * 2^{11}) * (2^{12})$

↓ Total no. of sectors. ↳ section size

= $2^4 * 2^1 * 2^9 * 2^{11} * 2^{12} = 2^{37}$ = 128 GB.

(ii) If speed = 3600 RPM and avg seek time = 30 msec. what is the disk I/O time.

Speed = 3600 RPM

Rotation time = $R = \left(\frac{1}{3600} * 60\right)$ sec = $\frac{1}{60}$ sec.

$LT = R/2 = \frac{1}{60 \times 2} = \frac{1}{120}$ sec. = 8.33 msec.

Section size = B = 4 KB.

Track size = $2^9 * 4$ KB

16.66 msec — $2^9 * 4$ KB

? — 4 KB

$\Rightarrow TT = \left(\frac{1}{60} * \frac{1}{2^9}\right)$ sec = 32.55 μsec

= 0.03255 msec

∴ Disk I/O time = ST + LT + TT

$= \left(30 + \frac{1}{120} + 0.03255\right)$ msec

$= (30 + 8.333 + 0.03255)$

= 38.36588 msec.

(iii) In one rotation the disk head can read a complete track

= All the sectors present in one track

= 512 * 4 KB

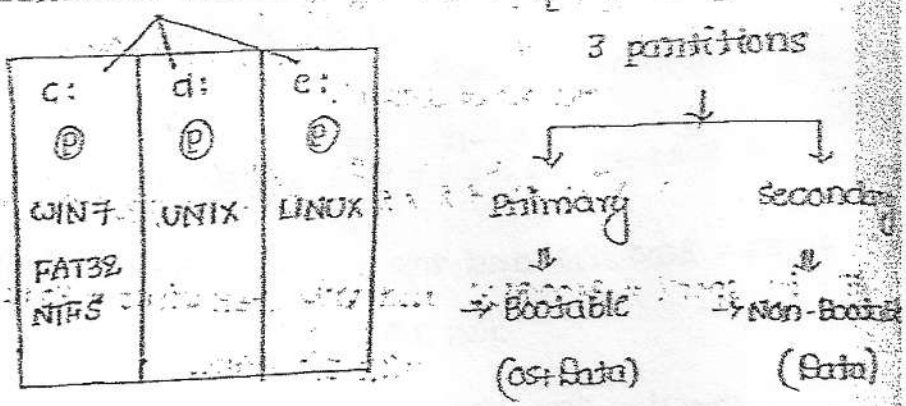
= 2 MB.

$R = \frac{1}{60}$ sec, it can transfer 2 MB data

in unit time it can transfer = (60 × 2) MB

∴ Data transfer rate = 120 MB

Logical structure of Disk (formatting process).



* Disk => Multiboot

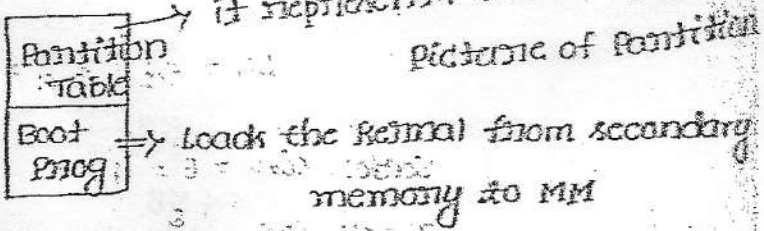
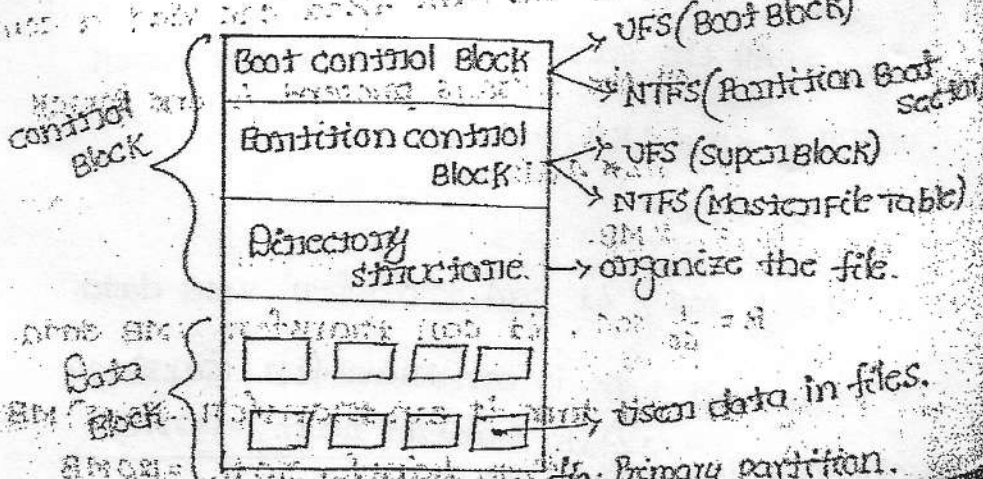


fig: MBR => Master Boot Record.

* In the partition of a system, there most be atleast one primary partition and the rest may (or) may not be seco
* computer system, which supports multiple os to boot is called "Multiboot computer"

Partition Structures:



Implementation Issues:

1. File Allocation Method / Disk space Allocation Methods:

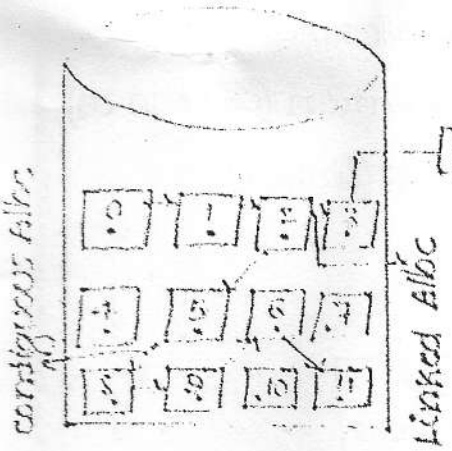


fig: logical view

[DB] : Disk Block on Disk space

→ The smallest unit of a file storage.

DBA : Disk Block Address (bits)

DBS : Disk Block size (B).

→ DBA + DBS together determines

∴ Max file size = Disk size.

Eg: DBA = 16-bits

DBS = 1KB (1 block = 1KB)

⇒ No. of Disk Block Addressable through the DBA = 2^{16}

$= 64K$

Maximum file size = DBA * DBS

= 64K * 1KB

= 64MB.

Eg: If it is not the multiplexer/decoder logic of combinational circuit. Here 1DBA can point to any 1 of total 64K address. But when in use at a time 1 DBA will point to one DB only. If a pointer to address a file size 64MB, and each DBA length of 16-bit. then maximum file size?

$\therefore$ DBA = 64K * 2B

$= 2^{16} * B$

DBS = 64MB $\Rightarrow 2^{26}$ B

Maximum file size = DBA * DBS $(2^{16} * 2^{26})$ B

File Allocation Methods:

1. Contiguous Allocation
2. Indexed Allocation (CG + Non-CG)
3. Noncontiguous (linked)

1. Contiguous Allocation:

Filename	start DBA	File size/ No. of blocks
Test.c	6	4 (3, 4, 5, 6 blocks)
ab.c	11	3 (7, 8, 9 blocks)
Prg.c	15	2 (0, 1 blocks)

2. Non contiguous Allocation:

→ Physical link is created by maintaining list type data structure (linked list)

Filename	start DBA	End DBA
Test.c	4	17
Prog.c	10	15
ab.c	8	12

Performance	Contiguous	Non contiguous
* Internal fragmentation	Yes	Yes
* External fragmentation	Yes	No
* Increasing the existing file size	inflexible	Flexible
* Type of access	Sequential & Random ⇓ fast	sequential ⇓ slow.

Drawbacks in Non contiguous:

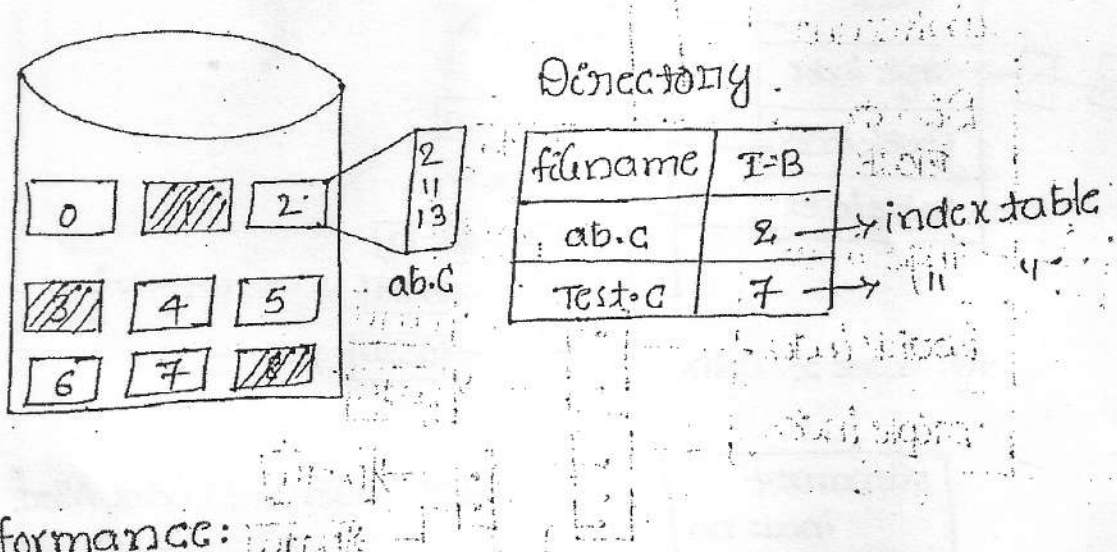
some disk space is consumed for storing pointer, that addresses next instruction block.

Vulnerability of links => breakage of links => Truncation.

Indexed file Allocation: (CG + NCG)

Each file is access with an individual block.

Index block contains all the block addresses (pointers).



Performance:

-> Internal fragmentation : yes

-> External fragmentation : NO.

-> Access Mode : Direct or Random

-> Maximum file size gets limited by index table size.

-> Lot of space is wasted maintaing the index table for small size.

Combined Allocation Scheme:

* To overcome the problems of static space + Maximum file size the combine scheme is used
* suppose, 15 pointers to be stored in an index table.

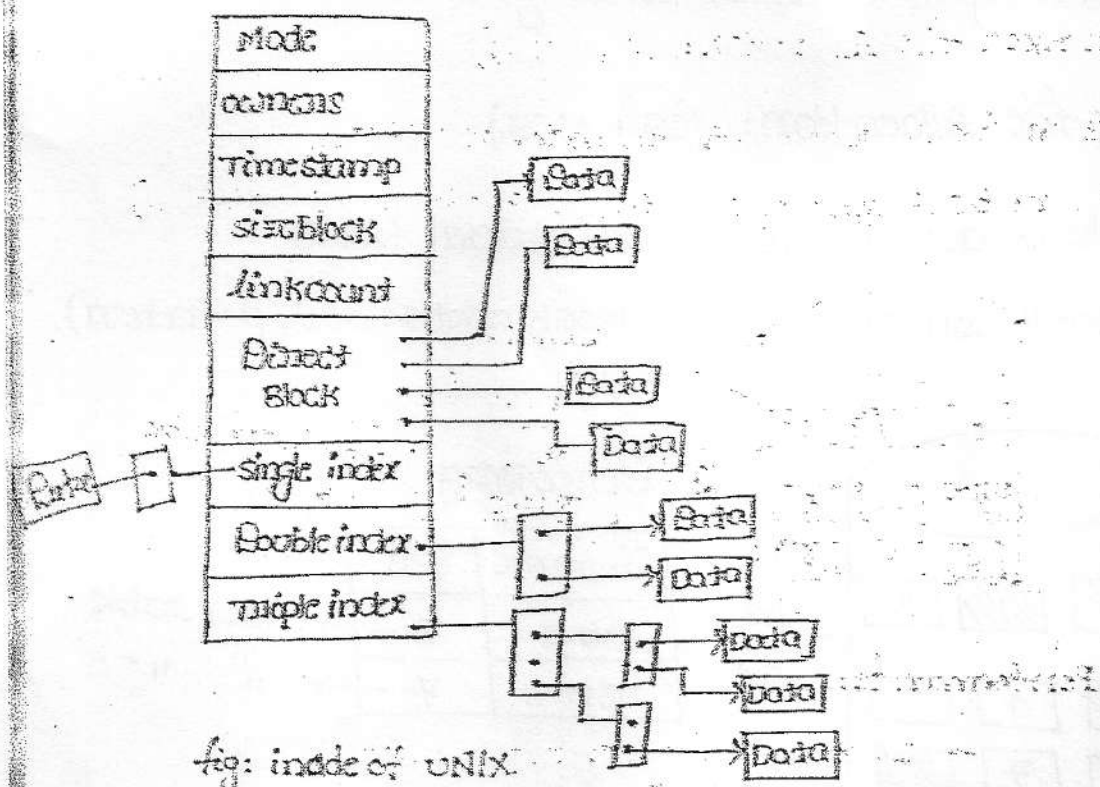
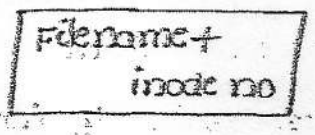


fig: inode of UNIX



Directory

Also called "Extended Index Alloc."

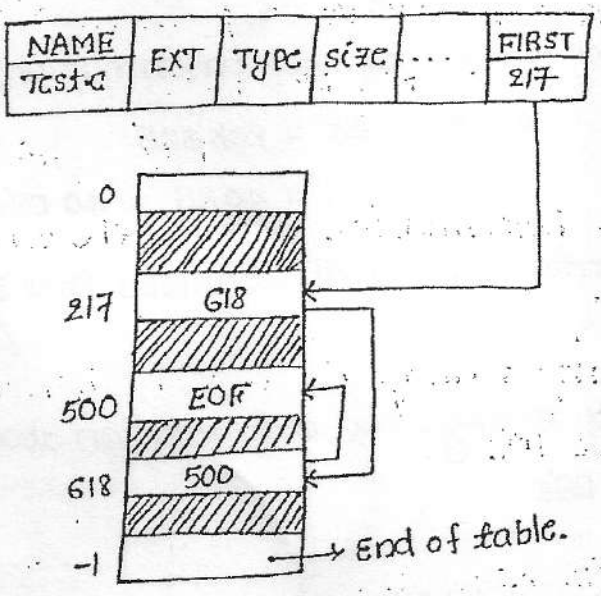
* The first 12 pointers are used as direct pointers which points to actual data blocks.
* The data for a small file would not require a seperate index block (upto 12 block size).
* The 13th pointer is used a single indirect pointer which points to one index block instead of data block.
* The 14th pointer is used as a double indirect pointer which points to a single indirect block, which inturn points several index blocks

Note: Although the largest file size possible = 138.25 GB.

The maximum file size supported by a system with 16-bit DBA and 1K DBS is only 64 MB.

File Allocation Table scheme:

* it is a modification over linked allocation.
* File data is distributed over several DB's as in linked allocation.
* The directory contains the general attributes of the file along with its first DBA or starting block address.
* A section of disk at the begining of each volume/partition is set aside for the File Allocation Table.



* The table has one entry for each disk block (DB) and it is indexed by the block number.
* Directory contains the file name and the block number of the first DB of the file.
* The table entry, indexed, by that block number, contains block number of the next block and so on.
* This chain continues until the last block of the file containing a special end-of-file character.

* Unused blocks are indicated by @ in the table.

Random access is improved, the disk head co
easily find the required block addresses from FAT

* A scheme with 32-bit pointer, known as FAT-32 is
employed by Dos

Free Disk-space Management:

Suppose we have a disk of size: 20 MB.

DBS: 1KB

DBA: 16-bit = 2B

→ Num of DBS = $\frac{20MB}{1KB}$

= 20K

→ space needed to keep information/addr
of 20K DB's = 20K * 2B

= 40KB = 40 DB's disk

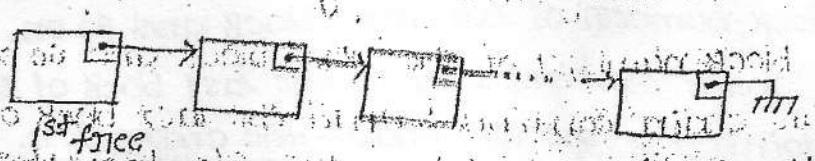
Goal: To keep the information about all the free DB's in
the Disk.

Criteria: Efficiency manage the information about
the free DB's.

1- Free Linked List Approach:

→ All the blocks are treated as nodes of linked list.

→ The first free block, is the head, it points to the n
free block, which in turn points to next free block, s
until the last free block.



→ In this scheme, in our example initial size of disk is 20 K and contains all the DBs.

Drawback:

→ Not efficient

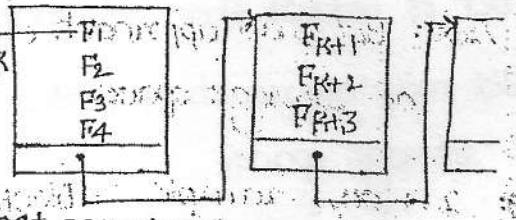
→ Access is sequential.

→ Disk I/O to locate each free disk and manipulate it will be very costly interms of CPU time.

2. Grouping of free list Approach:

→ Instead of maintaining a linked list structure, keep the free DBs addresses in groups, i.e in one block, keep the addresses of first K free blocks.

→ The last entry of that block points to the next block, which might be containing addresses of another K free blocks and so on.



Advantages:

→ searching time is greatly reduced almost constant

→ More efficient interms of I/O time compared to free list approach.

3. In our example 1 DB contains 1 KB space.

Address of each DB = 16-bit = 2 Bytes.

→ Each DB can hold address of $\left(\frac{1KB}{2B}\right)$ = 512 blocks

→ out of the 512 entries the last 1 should be pointing to the next block of free DBs but that will lead to more complex measurement.

Then, total 41 blocks are needed, last block : 40 entries.

∴ No. of blocks required to hold the address of all free blocks = $\frac{20K}{512} = \frac{20*2^{10}}{2^9} = 40$.

3. Bit Vector / Bit Map Approach:

→ The storage might be reduced further in this approach.

→ Use a bit to represent whether the block is free or not

0 – Block is free

1 – Block is in use.

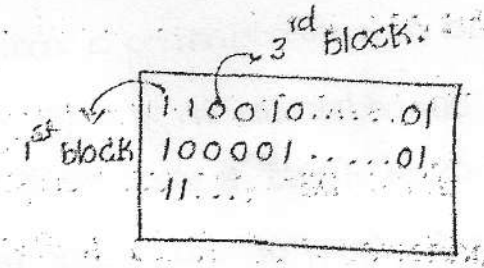


fig: Bit Map Block.

Drawback: Searching is more time consuming, free bit is better than bit map in this respect.

Adv: Bit map approach is better than free list in terms of storage space.

Eg: In our example 1 block size = 1KB = 8K bits.

⇒ Num. of blocks needed to hold the bit map of all the 20K blocks = $\left(\frac{20K}{8K}\right)$

= 2.5 ≅ 3 block are required.

4. Maintaining counter Approach:

→ In general OS allocates a series of blocks contiguously

→ The free blocks are also often found in a contiguous spa

→ Thus we can maintain a two entry information from each free "section".

(i). The address of the 1st free blocks.

(ii). No. of free blocks from the 1st free block (n_i).

i,e $<a_i, n_i> = <1, 10>$ starting from block 1, 10 block are free i,e (1, 2, 3, 4 ..... 10)

$= <50, 5>$ starting from block 50, 5 blocks free i,e (50, 51, 52, 53, 54)