26th June, 2010

Saturday

Operating Systems

Topics :

(1) Background

(2) Process management

* Process concepts
* CPU scheduling
* IPC & synchronisation
* concurrency
* Deadlocks
* Threads

(3) Memory management

* concepts
* RAM chips
* Techniques
* Virtual memory

(4) File system & I/o management

* Interface
* Device characteristics
* Implementation issues

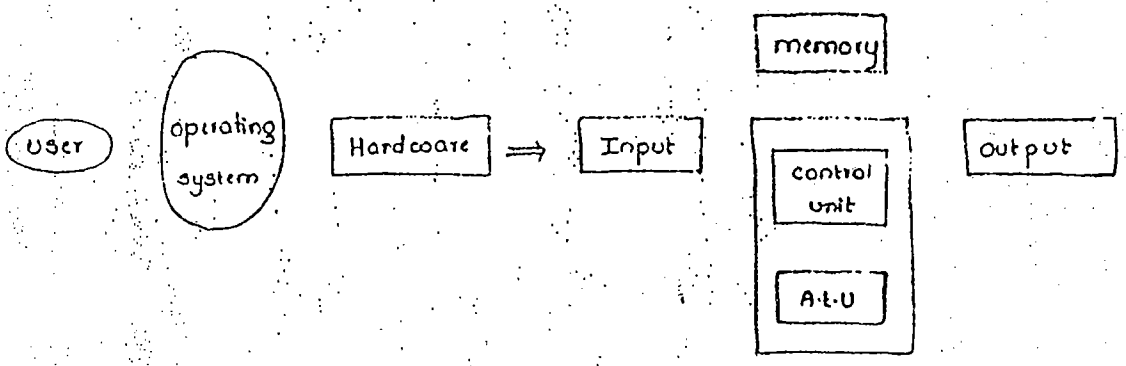
(5) Protection mechanisms

(Q) What is an operating system ?

Ans: Interface between user and Hardware

* Resource mandger
* control program(s)
* A set of utilities to simplify application development.
* Acts like a government.

Von- Neumann Architecture :



(i) Control signals

(ii) Mathematical operations (Data, Register)

clock cycles

c = a+b

load R_1 m[a]

load R_2 m[b]

Memory :

* Primary (main memory, physical ⇒ RAM, ROM, cache, Registers)
* Secondary (Auxiliary ⇒ HD | DVD | Pendrives)

* Processor cannot fetch data directly from secondary memory. Instead
- Instruction Register (IR) is used, from where instructions are fetched an
executed sequentially.

* If IR requires any operands, then it fetches the operands and
executes. as :-

* Fetch cycle
* Decode cycle
* Execute cycle
* Interrupt cycle

→ Instruction cycle.

* The communication among user and cpu can be represented by Von-Ne
-mann architecture.

Main Objective of operating system :

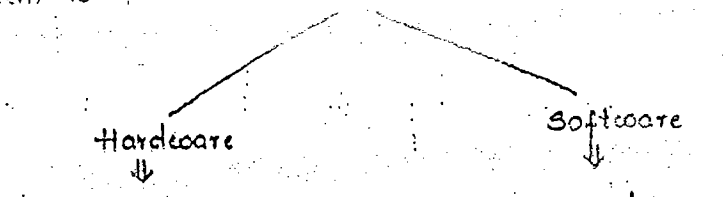
* It takes the complete control of Hardware, and create a platform, which
is easy to user to write different applications.

Command Interpreter :

* A program that interprets the commands of the user.
* It acts as Interface between the user and Kernel.
* There are of two types :
  * Text based O.S (shell) i. DOS, UNIX, NETWARE.
  * GUI based O.S (Interface) i. WINDOWS, UNIX.

* Operating system is a set of system programs.

Operating System is visualised as resources.



* CPU
* memory
* I/O

* Device drivers
* semaphores
* messages
* monitors.

Functions & Goals :

Goals :

(1) convenience (user friendly)

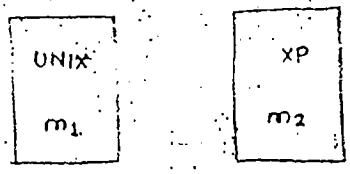
(2) Efficiency (resource utility)

(3) Reliability & Robustness

(4) Scalability (Ability to evolve)

(5) portability (Different platforms)

Consider an example :



Execution time : 50 sec 10 sec

To find a particular content in "UNIX" O.S, we must write it's 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 in "XP" O.S, it is an easy process that any user can

approach through (i.e. clicking "start" button & then "search")

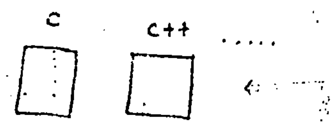
* It takes more time than unix to execute, but it is more familiar to all kinds of user to operate.

* So, "convenience" is more important than "efficiency".

* In some real time systems, "Efficiency" is more important than "convenience".

First Generation — No operating system — Card-Readers, Punch cards; manual Intervention.

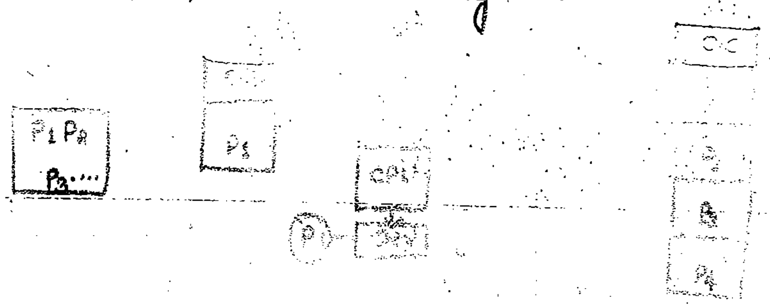
Second Generation — Magnetic Tapes — Batch processing ⇓ Clubbing similar programs at one place & processed one at a time.



Third Generation — Hard Disk Technology

- Uniprogramming ↓ (DOS) main memory
- multiprogramming ↓ (UNIX, XP)

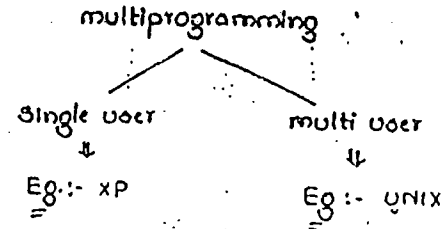
*(coastage of idleness of CPU)



Multiprogramming:

* Ability of operating system to hold multiple ready to run programs the main memory so that if the running program requires I/o, the CPU can be switched to another ready program.

* Multiplexing of CPU among ready programs in memory.



Non-Pre-emptive:

* Running process is released voluntarily (on it's own) as follows:

    * Completion of process

    * I/o request.

Pre-emptive:

* Running process can be forced to released based on:

    * Priority

    * Time sharing

Time sharing systems (pre-emptive):

Eg:- windows XP

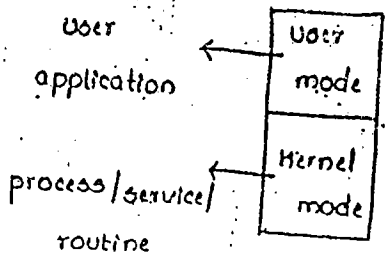
first version of windows: 3.0, 3.1, 3.11 } non-preemptive

Architecture of multiprocess support:

(1) Direct memory Access (DMA) ⇒ Since computation & I/O required, we need DMA.

(2) Address Translation

(3) Atleast two modes of CPU execution.



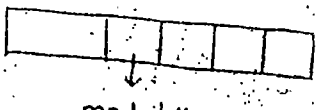
non-priveleged (pre-emptive)

system/supervisory/privileged mode (non-pre-emptive)

* In Kernel mode, the process have atomic execution (without pre-emption)
* In user mode, the process have non-atomic execution (with pre-emption)

Program status coord (PSW):

It represents the mode, in which the user operates in.



0 - user mode
1 - Kernel mode

System call Vs Library call:

```
main()
{
    int a,b,c;
    scanf("%d %d", &a, &b);
    c = a+b;
}
```

scanf → (Library call) → (BSA) ⇒ Branch & save user mode Address (BSA)

Stored in std.

* fork() creates child process and the [illegible] for [illegible] because it is a part of OS and the conversion takes place (from user mode to system mode)

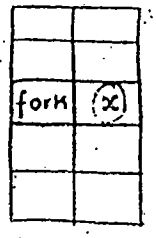
* Execution of supervisory call (SVC) involves the software interrupt, which handles Interrupt Service Routine (ISR).

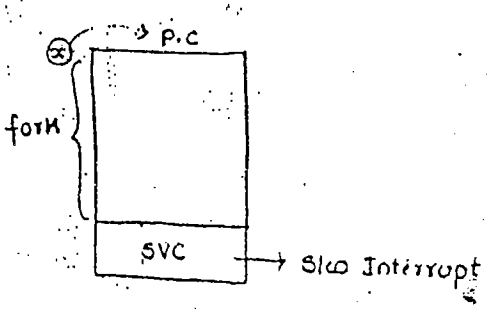
* When a SVC is generated, ISR changes to 1 (non-preemptive mode). Then after the completion of all the functions in Kernel mode, again an SVC is generated, where ISR value is changed from 1 to 0 (pre-emptive mode)

System call Interface (SCI) ⇒ UNIX

In windows 8.8, SCI is considered as API.

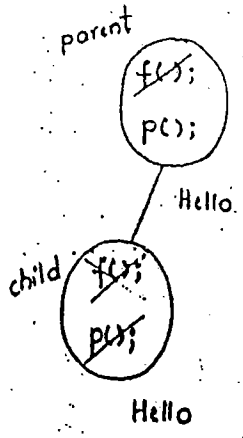
ISR ⇒ changes mode bit & Dispatch the table



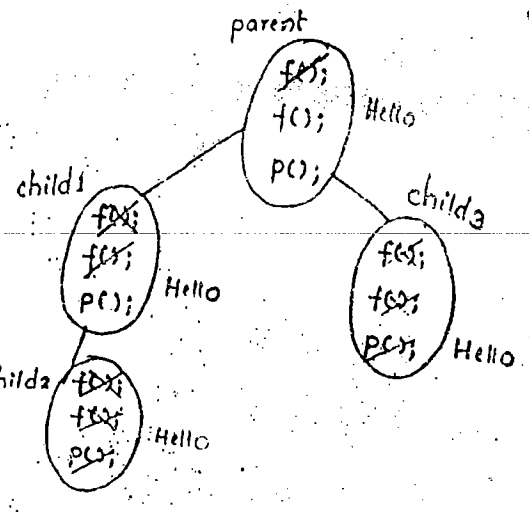


1)
```
main()
{
    fork();
    printf("Hello");
}
```

Output : Hello
Hello



(a) main()
{
fork();
fork();
printf("Hello");
}



* If 1 fork ⇒ 2 prints
2 forks ⇒ 4 prints
3 forks ⇒ 8 prints
⋮
n forks ⇒ 2^n = Total

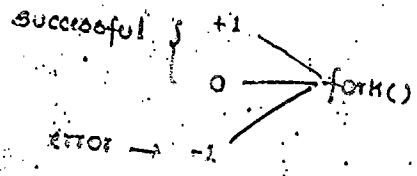
output : Hello
Hello
Hello
Hello.

$n_{child}^{reco} = 2^n - 1$, parent = 1

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

(a) n-1 (b) n-1 (c) n^2-1 (d) 2^n-1 (reco forks)

* Any system call may return either +1, 0, -1 values.



child parent error

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

```
id=fork();
if(id==0)
{ }
else if(id>0)
{ } ;
```

child id [0]

```
id = fork();
if(id==0)
{ }
else if( id>0)
{ }
```

Difference between Kernel mode & user mode :

Compilers, editors and similar application-independent programs are not part of the O.S, even though they are typically supplied by computer architecturer. This is crucial, but subtle point. The O.S system is that portion of the software that runs in Kernel mode. It is protected from user tampering by hardware.

Compilers & editors runs in user mode. If a user doesn't like a particular compiler, he/she is free to write their own clock interrupt handler, which is a part of O.S.

01st July, 2010

Thursday

Process concepts

Program Vs Process :

* Program under execution.
* Unit of execution
* Schedulable / Dispatchable unit
* Instance of program
* Central locus of control (cos)
* Animated spirit

Process : Program

1 : 1

1 : multiple

* Executing program resides in secondary memory.

Program	Process
* In secondary memory	* In main memory
* Without resources	* Utilization of resources
* Passive	* Active (Alive)

Formal definition of process :

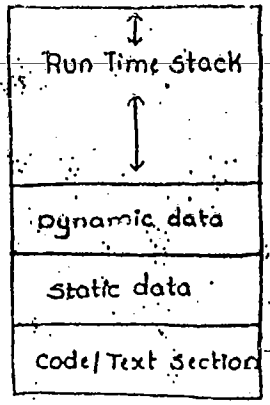
* Developer / Designer views process as a simple Datastructure.
* Every datastructure is always defined with four parameters :
  * Definition
  * Representation (process structure)
  * Operations → Implementation
  * Attributes (public, private)

Process Structure (Representation) Implementation in main memory

Abstract view of process:

* program consists of data and instructions.

Data
- Static ⇓ : variables, (Data structures whose sizes are known in prior)
- Dynamic ⇓ (space allocation during run-time using Dynamic memory Allocation (DMA))



(DMA). (name of memory area) Eg: Heap
(name of memory) Eg:- stack
(programs)

Run Time stack ⇒ Activation records are maintained wherever function calls are generated.
⇓
Every process maintains this stack
For storing addresses (recursive) we use this stack.

Operations :-

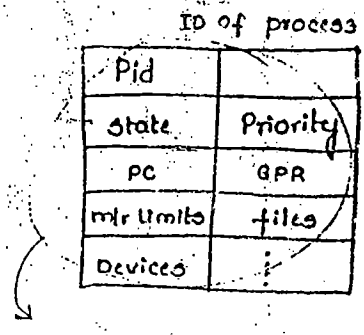
* Essential resources required for stopping the execution of process, then then some resource utilities are used.
* When a program is loaded in main memory, resources are allocated and then CPU, schedules all the process.
  * create (resource allocation)
  * Scheduled (CPU)

* Execution (CPU)
* Blocked (I/O)
* Resume
* Suspend( )

Attributes :-

* Process Id (pid)
* Priorities
* Process state
* Program counter
* memory limits
* list of files
* list of devices
* Type & size (in bytes)
* Protection.

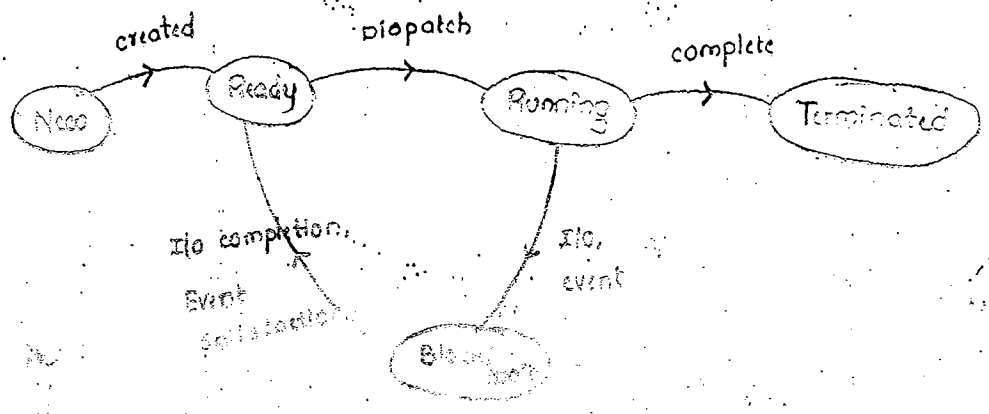
Process control Block (PCB)



process context / process environment.

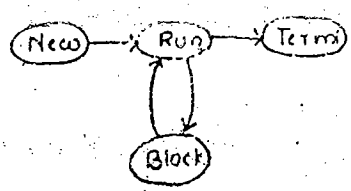
* All the process attributes are stored within the process control block (PCB

** Process State :-



Above diagram represents :

(a) Uniprogramming (no "ready" is present)

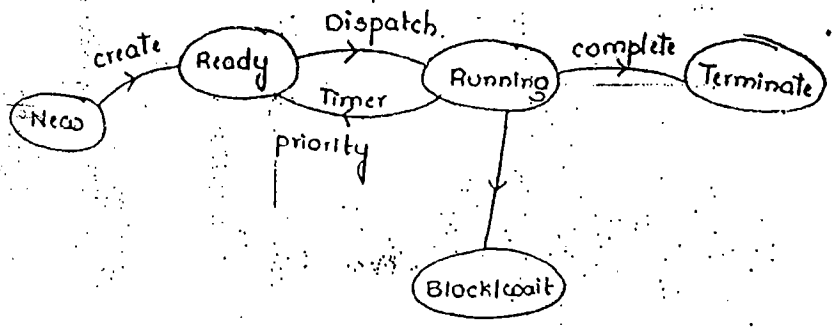


(b) pre-emptive multiprogramming

(c) non-preemptive multiprogramming

(d) multiple cpu based o.s

Pre-emptive multiprogramming :-



Degree of multiprogramming = no. of processes

Degree ↑, when process ↑, so, we suspend process whenever required.

process suspension ⇒ swapped out on temporary basis and later on resumed.

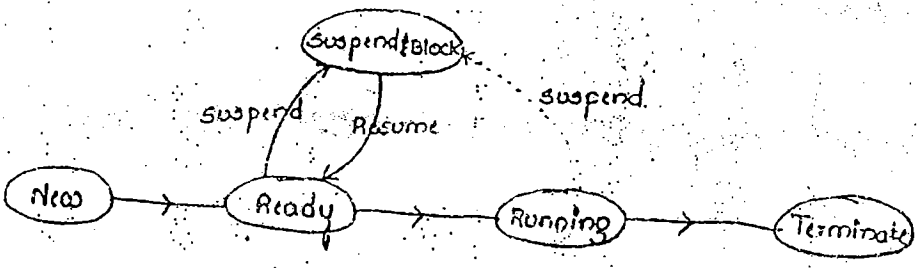
There is no suspension for the following process states : ↑ performance

* New
* Terminate

Most desirable state for suspension is : "Ready" state.

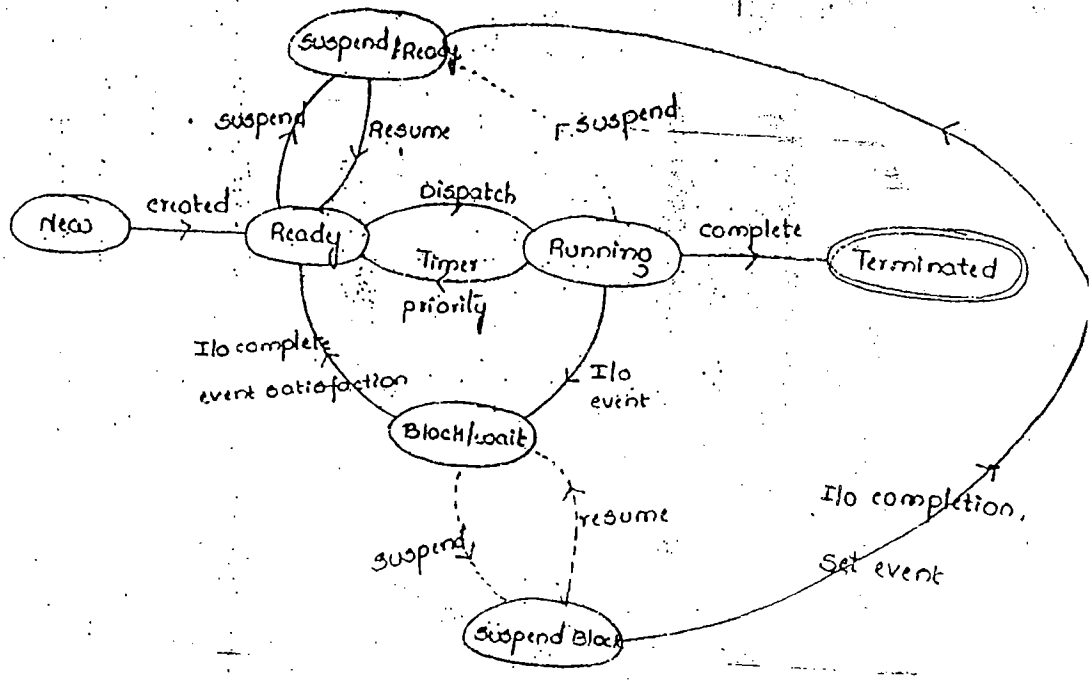
Because, it is waiting for I/o or other events, so, when suspended it get confused.

If "Ready" process is suspended, then it moves to "suspend" state.



Block/wait

When "Ready" state is suspended.



* Only Ready, running, Block states ⇒ can be suspended
* process waiting for Ilo operations are not suspended (dangerous). If get suspended, it moves to suspend block state.
* Suspension of running process are not preferred always. If suspended, it releases the resource preemptively.
* So, Ready [illegible] temporarily stored in secondary memory.

(Q) Consider a system with n CPUs and m processes where n, and m ($n \geq 1$, $m \geq n$) calculate lower bound and upper bound on the process states.

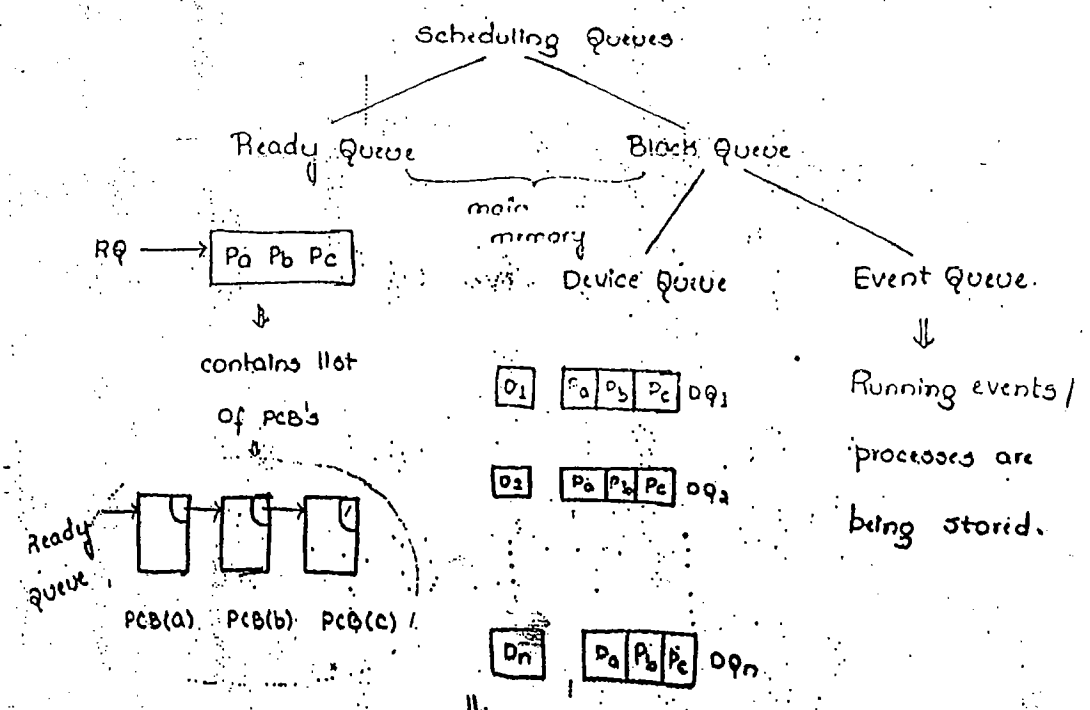
* Ready
* Running &
* Block states:

Ans:- maximum Ready state processes are = m

maximum running state processes are = n.

	Min LB	Max UB.
Ready	0	m
Running	0	n
Block	0	m

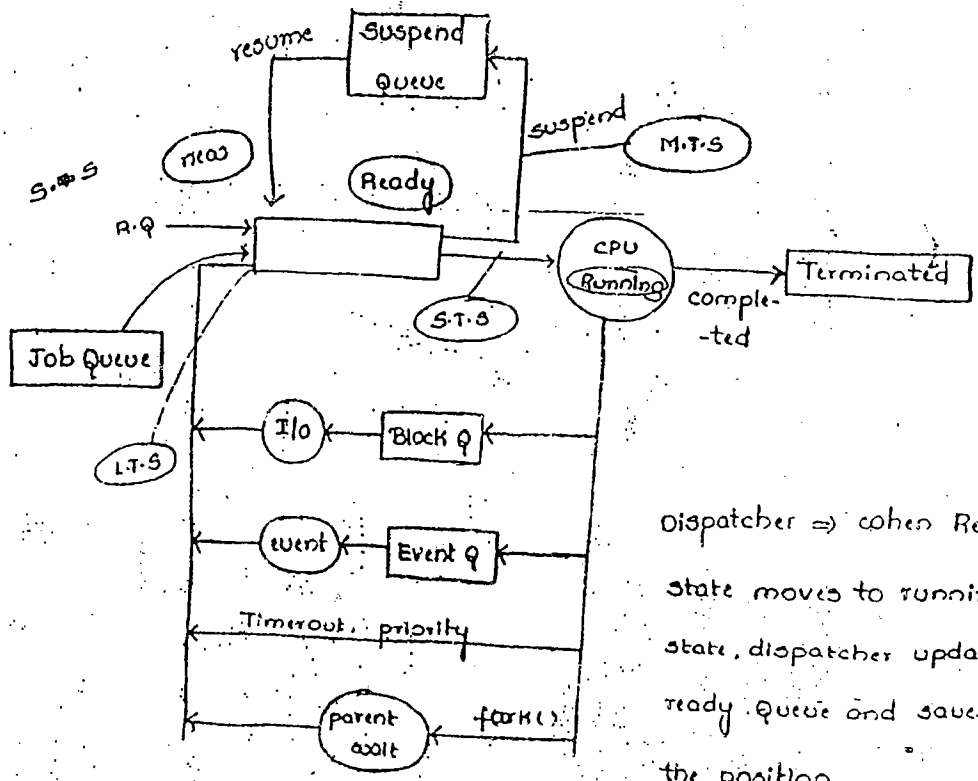
Scheduling Queues & State-Queing Diagram :-



Scheduling Queues — Suspend Queue ⟹ It contains suspended proc

Secondary memory — Job Queue ⟹ It contains PCB's of Job Qu

Process State Queue diagram :-



Dispatcher ⇒ when Ready state moves to running state, dispatcher updates ready Queue and saves the position.

Long term Scheduler - loading new programs from secondary to main memor

Job Queue

Short term scheduler(CPU Scheduler) :-

It decides that which process (ready process) should run next.

middle term scheduler (suspended Queue)

(It decides that which process should be suspend(ed) ...)

Saving and loading the PCB from one process to another is called "context switching".

* "context switching" has directly impact on volume of information in PCB.

multiprogramming ⇒ long term scheduler.

CPU Scheduling

CPU scheduler :

* Decision of which process to run is taken.

Goals :-

* Maximize CPU utility
* Minimize Response time, waiting time.

Process Times :

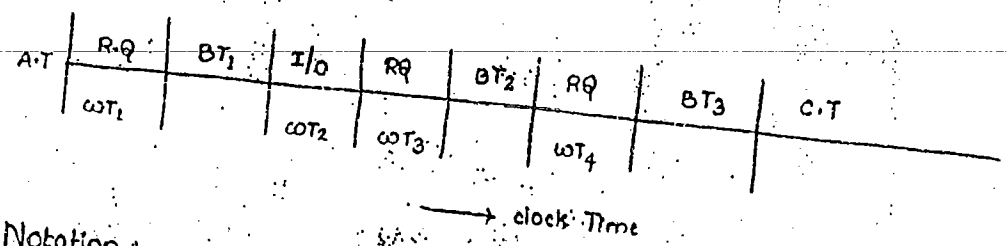
* Arrival Time (A.T) - Submission time
* Waiting Time (Ready Queue / BQ)
* Burst Time (CPU) - Service Time (B.T)
* Completion Time (C.T)
* Turn Around Time (TAT) = completion Time (C.T) - Arrival Time (A.T)

Response Time (R.T) :

* Time of submission of request by a process, to obtain a result / response (first response)

Deadline (D):

* No. of processes completed within the deadline.



Notation:

(1) n-processes ($P_1 \ldots\ldots P_n$)

(a) A·T (P_i) = A_i

(3) B·T (P_i) = X_i

(4) C·T (P_i) = C_i

(5) Deadline (P_i) = D_i

(2)

Formulae:

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

(b) WT · TAT (P_i) = $\frac{C_i - A_i}{X_i}$

(c) Avg · TAT (P_i) $_{i=1,\ldots\ldots n}$ = $\frac{1}{n} \sum_{i=1}^{n} (C_i - A_i)$

(d) W·T (P_i) = TAT − BT

= $C_i - A_i - X_i$

(e) Avg. W·T (P_i) $_{i=1\ldots n}$ = $\frac{1}{n} \sum_{i=1}^{n} (C_i - A_i - X_i)$

(f) Schedule length (L) = max (Ci) − min (Ai)

(g) Throughput (μ) = $\frac{n}{L}$

b) Deadline overrun = $C_i - D_i$

If $(C_i - D_i) < 0$ underrun

If $(C_i - D_i) = 0$ on deadline

If $(C_i - D_i) > 0$ overrun.

Schedule :-

n-processes. n=3. (P_1, P_2, P_3)

$|P_1|P_2|P_3|$ ⟷ length
$c_1\ c_2\ c_3$

$|P_2|P_3|P_1|$ ⟷ length
$c_2\ c_3\ c_1$

07/07/2010

Wednesday

CPU scheduling Techniques

- Non-preemptive
- pre-emptive

(1) First come First Serve (FCFS) :

Criteria : Arrival Time (process)

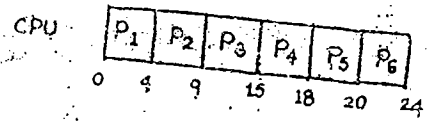
mode : non-preemptive

TAT = CT - AT

WT = TAT - BT

P.NO	A.T	B.T (CPU)	CT	TAT	WT
1	0	4	4	4	0
2	1	5	9	8	3
3	2	6	15	13	7
4	3	8	18	15	12
5	4	9	20	16	14
6	5	4	24	19	15

Gantt chart :



Arrival Time of $P_1 = 0$

i.e., at '0' clock time P_1 arrives

P_2 C.T = P_1 C.T + P_2 B.T

Completion Time (P_n) = Completion Time (P_{n-}) + Burst Time (P_n)

Eg. 2 :-

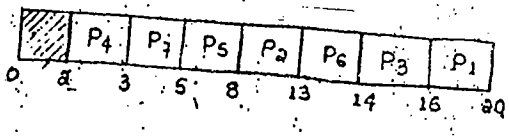
PNO	AT	BT	CT	TAT	WT
1	8	4	20	12	8
2	5	5	13	8	3
3	7	2	16	9	7
4	2	1	3	1	0
5	4	3	8	4	1
6	6	1	14	8	7
7	3	2	6	2	0

18

TAT = CT - AT

WT = TAT - BT

Gantt chart :



$L = 20 - 2 = 18$

L = max. of C_i - min. of A_i

= 20 - 2

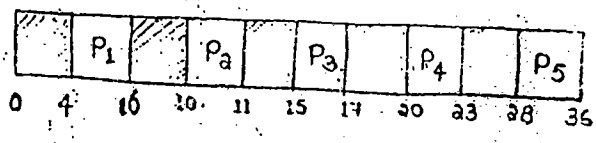
= 18

Since, there are no processes arriving at clock time = '0'. The process 'P_4' arrives at '2' clk time. So, the time (0-2) is subtracted from the total completion time. And, since 'P_4' arrives earlier than that of process 'P_7', ...

Process P_4 is considered first and then the other processes arrived within the particular Burst time.

Eg 3 :-

PNO	AT	BT
1	4	2
2	10	1
3	15	2
4	20	3
5	23	8
		16



WT(P) = 0

L = 36-4 = 32

Shortest Job First (SJF) :

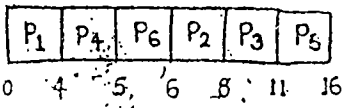
* It is also called as Shortest process next (SPN).

Criteria : Burst Time

Mode of working : Non-preemptive (default).

It also works under pre-emptive mode.

PNO	AT	BT
1	0	4
2	1	2
3	2	3
4	3	1
5	4	5
6	5	1



Initially, Process P_1 is ready for execution since it has arrival time = 0. so, it has been processed.

for processing P_1, it requires 4 clock cycles (B.T = 4), within the time, process P_2, P_3, P_4, P_5 are also ready for execution. within those processes, select the process which have less

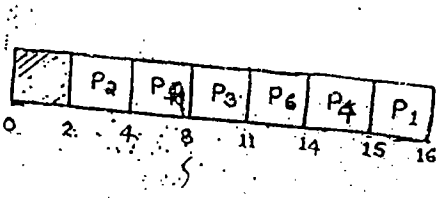
Burst time and process it. Here, P_4 has a B.T = 1. So, it is selected.

∴ Total clock cycles = 4+1 = 5. At the time of 6th clock cycle, process, 'P_6' (new) is available on ready queue. Since, the B.T = 1 of process 'P_6'. after completion of 'P_4', 'P_6' is get executed, eventhough it added recently in queue.

Eg. 2 :-

PNO	AT	BT
1	5	11
2	2	2
3	3	3
4	2	4
5	4	1
6	3	3

14



16 - 2 = 14

(3) Shortest Remaining Time First (SRTF):-

In pre-emptive mode, SJF is known as shortest Remaining Time first.

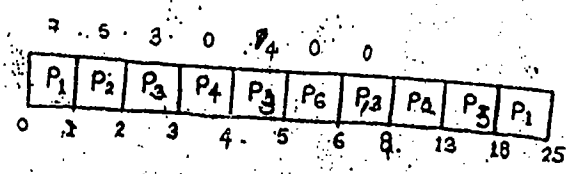
"Pre-emption of running process is based on the arrival of a new shorter process"

Criteria : Burst Time

mode : pre-emptive.

Eg:-

PNO	AT	BT
1	0	8
2	1	6
3	2	4
4	3	1
5	4	6
6	5	1

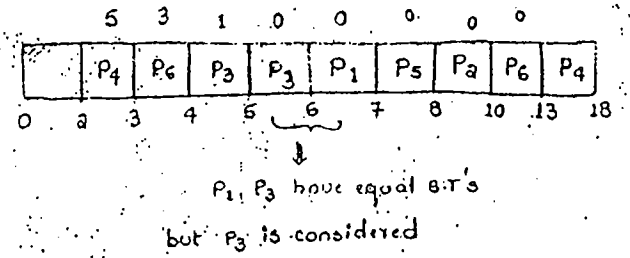


TAT (P3) = 8 - 2 = 6.

Initially, process P_1 has arrived, and within 1 clock cycle time P_2 arrives, then P_3 & so on. So, only 1 clock cycle is considered for P_1 within B.T = 8, so the remaining B.T = 7 remains. In P_2, B.T = 5, P_3 ⇒ B.T = 3, P_4 ⇒ B.T = 0.

At clock cycle = 4, process P_5 also arrives, but if we consider it, it has more Burst time than that of Process P_3. So, we consider 'P_3' as it is SRTF. Next, within 5 clock cycles 'P_6' also arrives, which have B.T = 1, so we consider 'P_6' after 'P_3'. and so on..... upto the end.

PNO	AT	BT
1	5	1
2	7	2
3	4	2
4	2	6
5	6	1
6	3	4



Performance of Shortest Job first (SJF):

Burst Time

Advantages:

* Enhances Throughput
* Average W.T & TAT reduces.
* It is implementable, with predicted / Estimated Burst time's.

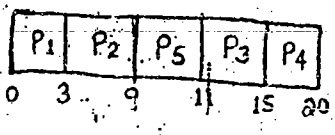
Disadvantage:

* Starvation to longest job.
* It is non-implementable, because burst time's of processes are not mentioned.

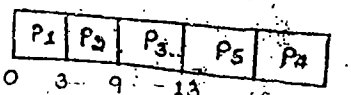
Non-preemptive SJF :-

P.NO	AT	BT
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

NP-SJF :



HRRN :-



$RR_3 = \frac{5+4}{4} = 9/4$

$RR_4 = \frac{3+6}{5} = 8/5$

$RR_5 = \frac{1+2}{2} = 3/2.$

$RR_4 = \frac{7+5}{5} = 12/5$

$RR_5 = \frac{5+2}{2} = 7/2$ (high)

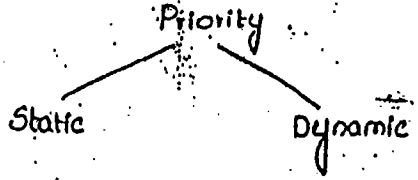
* process having high R.R, it is first considered.

* At clock cycle 9, Process P_5 is in Ready Queue, which have very little B.T.s. Process 'P_3' executed instead of P_5, because of long time waiting. This scheduling algorithm gives importance to long waiting processes as well as B.T processes.

(5) Priority based scheduling :

Criteria : priority

Mode : non-preemptive



Increasing the priority level regularly at the runtime is called as "Aging Algorithm".

Drawback :

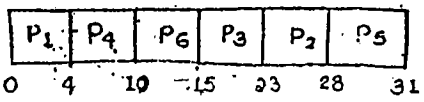
Starvation.

Higher Integer high priority

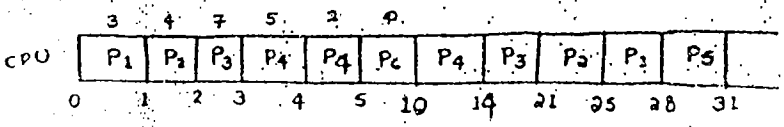
Eg:-

Priority	PNO	AT	BT
4	1	0	4
5	2	1	5
6	3	2	8
8	4	3	6
2	5	4	3
10	6	5	5

NP- priority :-

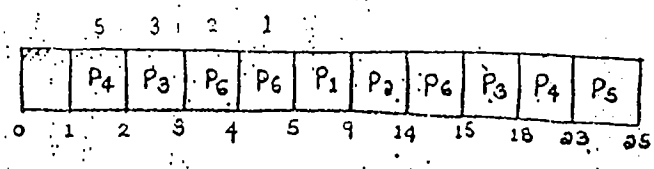


preemptive priority :-



a:

Prio.	PNO	AT	BT
12	1	5	4
9	2	6	5
7	3	2	4
6	4	1	6
4	5	4	2
8	6	3	3



Round - Robin (m.p / Time sharing) :

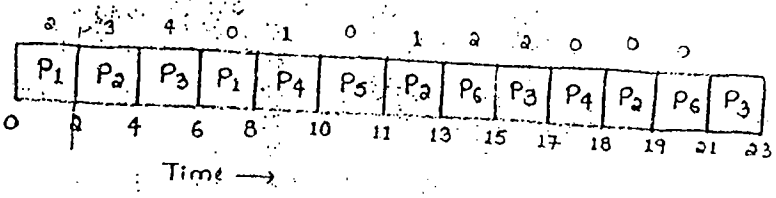
* Criteria : A.T + Time Quantum (TQ) slice

mode : pre-emptive

PNO	AT	BT
1	0	4
2	1	5
3	2	6
4	3	3
5	4	1
6	5	4

TQ = 2.

R.Q : P1 P2 P3 P1 P4 P5 P2 P6 P3 P4 P2 P6 P3



* Initially, P_1 arrives and executes till B.T = 2. (since Time slice = 2). Then, within 2 clock cycles, processes P_2, P_3 also arrives, then considered their B.T at TQ = 2 and hence P_1 requires another 2 clk cycles it must again be maintained in Ready Queue.

* At time = 8, all the processes arrives, based on their b.T's ready Queue is maintained.

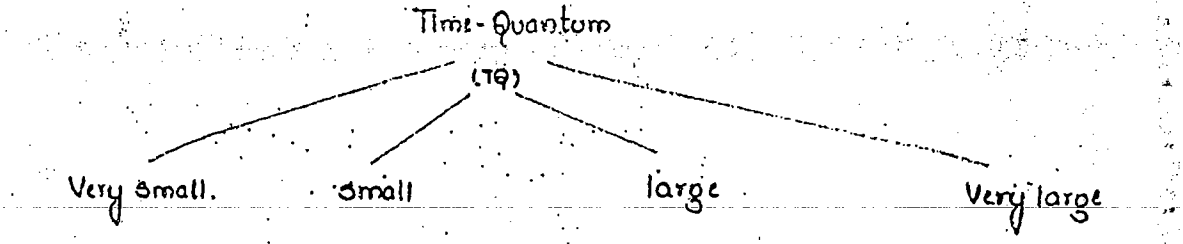
* If TQ is very small, efficiency (μ) = 0.

* If TQ is low ⇒ context switching occurs.

* If TQ is high ⇒ FCFS approach is followed

} so TQ value must be choosen moderately.

Performance of Round Robin

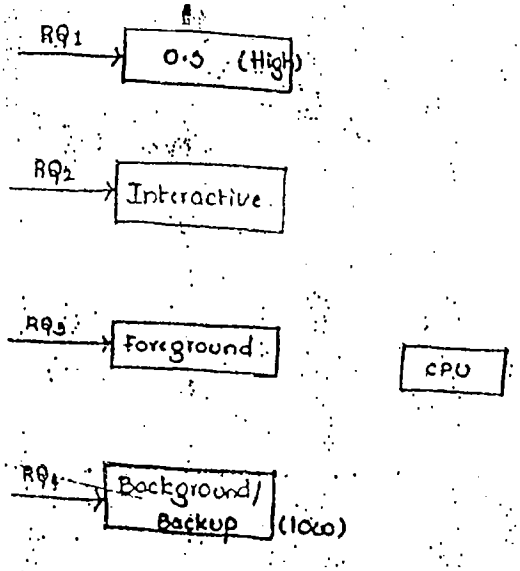


Efficiency (μ) ≈ 0

* more context switching (overhead)
* Improves response Time
* less context switching overhead
* less interative response
* works like FCFS approx
* Very poor response time

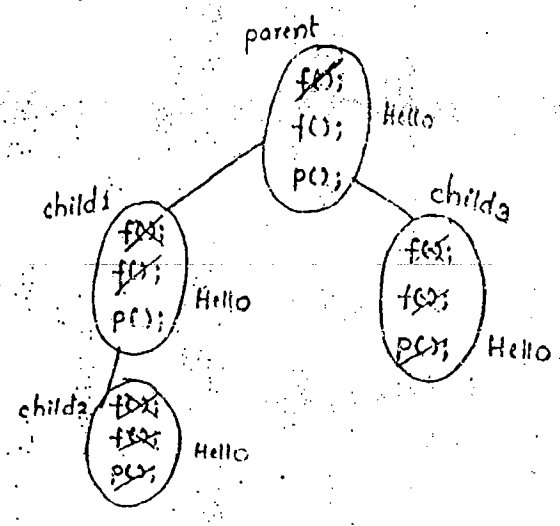
08/07/2020
Thursday

Multi-level Queue Scheduling



* In multi-level queue, the drawback is that, the second queue is serviced only when first queue is empty / complete.

(a) main()
{
fork();
fork();
printf("Hello");
}



* If 1 fork ⇒ 2 prints
2 forks ⇒ 4 prints
3 forks ⇒ 8 prints
⋮
n forks ⇒ 2^n = Total

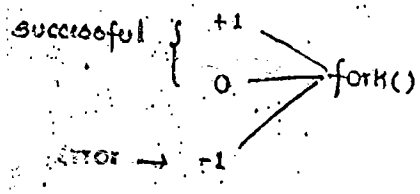
$2^n - 1$, parent = 1
(child)

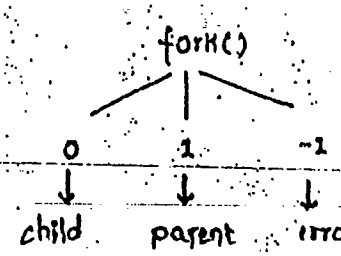
Output :
Hello
Hello
Hello
Hello.

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

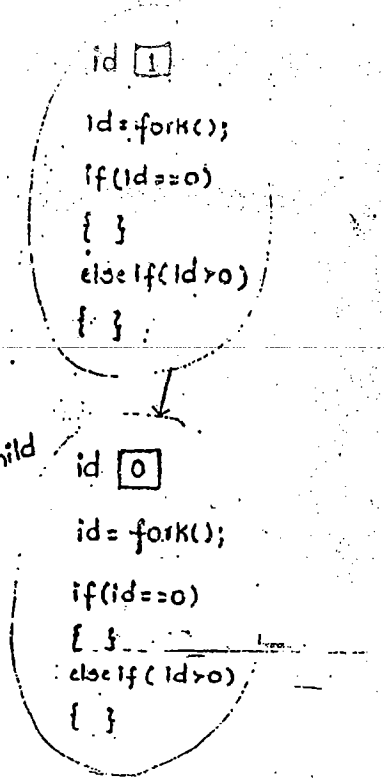
(a) n-1 (b) n-1 (c) n^2-1 (d) 2^n-1 (once forks)

* Any system call may return either +1, 0, -1 values.





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



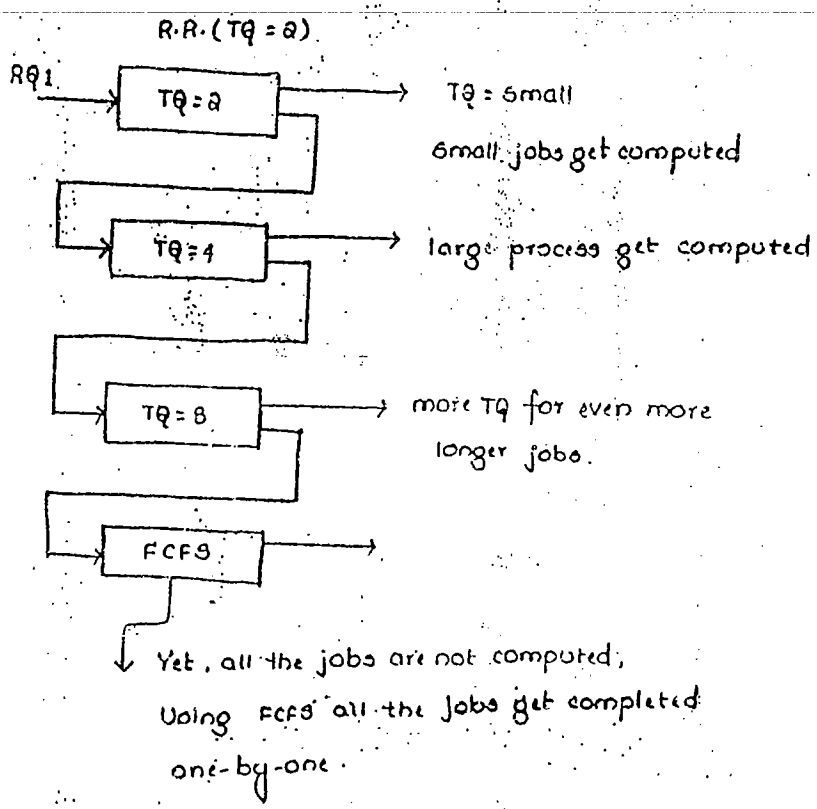
Difference between Kernel mode & user mode :

Compilers, editors and similar application-independent programs are not part of the O.S, even though they are typically supplied by computer architecturer. This is crucial, but subtle point. The O.S system is that portion of the software that runs in Kernel mode: It is protected from user tampering by hardware.

compilers & editors runs in user mode. If a user doesn't like a particular compiler, he/she is free to write their own clock interrupt handler, which is a part of O.S.

* Here, the feedback is pre-empted to the same Queue.

* So, we have multi-level feedback Queue to overcome this drawback.

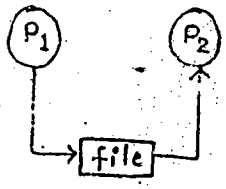


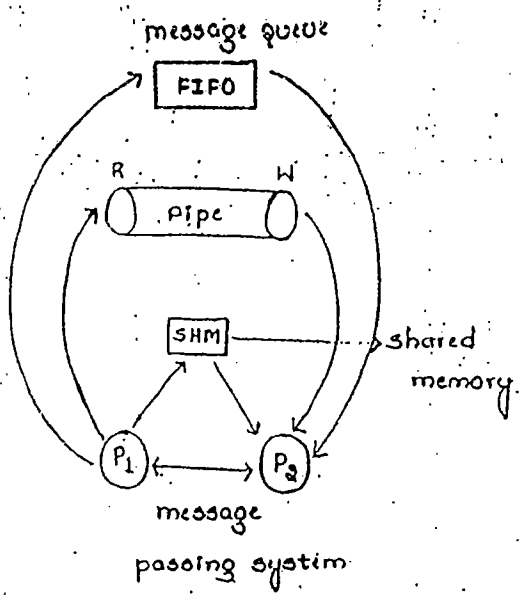
Inter Process communication & Synchronization:-

Inter process communication (I.P.C) :-

* For the communication among any two process, there must be a media to communicate.

Eg :- File (It is not always used)

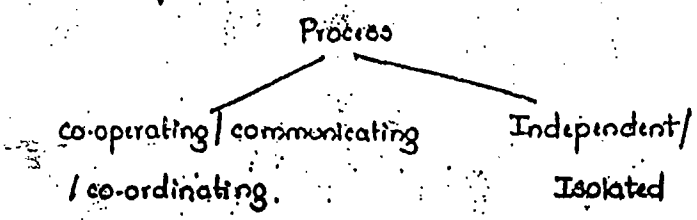




Problems due to lack of synchronisation :-

* Loss of data
* Inconsistency (wrong results)
* Dead locks.

How to achieve synchronisation?

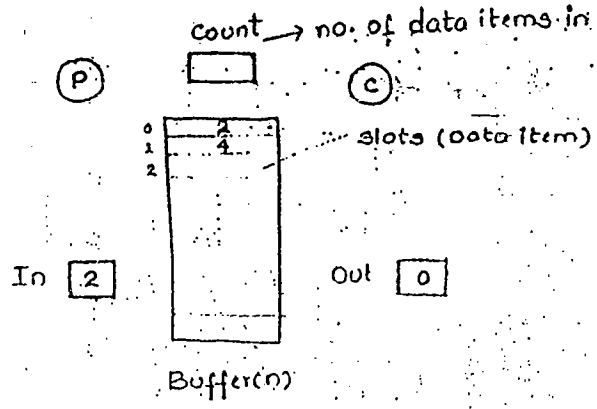


Co-operating process :-

* Two (or) more processes are said to be co-operative, if they get affected (or) affects the execution of other process.

* Otherwise, they are said to be Independant.

Producer - Consumer problem :-



* If Buffer is full, producer get affected.
* If Buffer is empty, consumer get affected.

```
#define n 100
int Buffer[n];
void producer(void)
{
    int itemp, in=0;
    while(1)
    {
        ProduceItem(itemp);
        while(count==n); // Busy wait
        Buffer[in] = itemp;
```

```
            In = (In+1) mod n;
            count = count + 1;
        }
    }
void consumer(void)
{
    int itemc, out = 0;
    while(1)
    {
        while(count == 0);
        itemc = Buffer[out];
        out = (out+1) mod n;
        count = count - 1;
        processitem(itemc);
    }
}
```

Lack of Synchronisation in Producer-consumer problem :-

Inconsistency :-

In producer,

```
count = count + 1 ;     // This step needs to fetch
(1) load Rp, m[count]       the variable "count" and
(2) Inc Rp                  then perform the operation.
(3) Store m[count], Rp
```

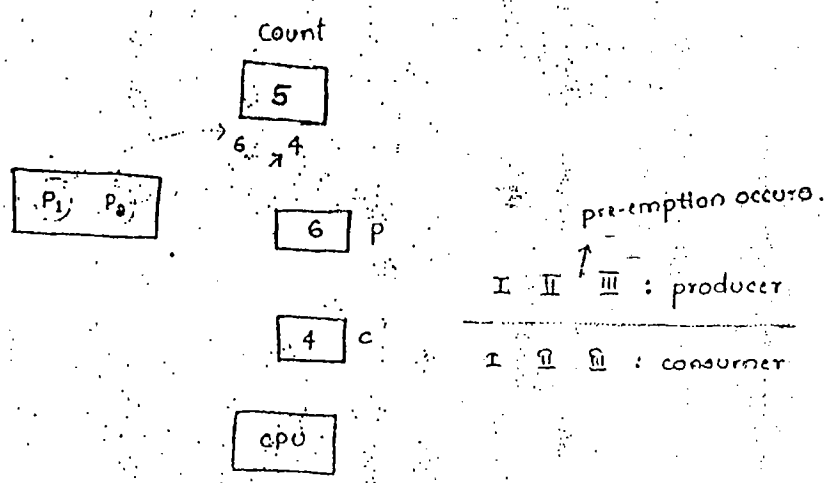
In consumer,

count = count - 1 ;

(1) Load Rc, m[count]

(2) Dec Rc

(3) Store m[count], Rc



* In producer,

while first tcoo steps get executed, pre-emption occurs. i.e.,

consider the count = 5 (initially). In step1, the value is loaded into registe 'Rp', and then in step2, the value in the register get incremented. so, the value of count = 6.

* Before storing the "count" value from the register, it got pre-empted. so, the producer have the count value = 6.

* In consumer,

count value = 5 is stored in a register 'Rc', and then Decremented. Before transferring the value from 'Rc' it get pre-empted. so it have count = 4. So, there is inconsistency, because count value is never equal to 6.

13/07/2010

Tuesday

Synchronization Mechanisms

Producer - Consumer :

Lack of synchronisation :

* Inconsistency
* Loss of data
* Deadlocks

Terminology :

1) Critical Section( ) & non-critical section( ) :-

* Critical section is that part of a program where shared resources are accessed.
* Non-critical section is that part of a program where no shared resources are accessed.

eg => shared resource (any variable/device)

2) Race conditions (concurrency) :

processes racing to get into critical section.

3) pre-emption :

A process get executed in the middle of other process.

count = count + 1

(1) load

(2) Inc

(3) store.

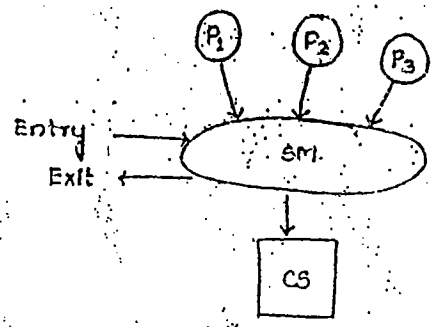
4) mutual exclusion:

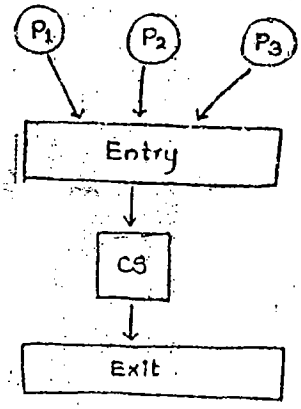
No two processes may be present in the critical section at same time.

Architecture / model of Synchronisation mechanism :-

Synchronisation mechanism :

* To ensure mutual exclusion so that there is no problem of Race condition among processes.



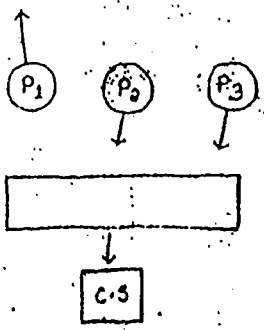


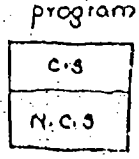
Requirements of Synchronisation mechanism :-

(1) mutual exclusion must always be guaranteed.

(2) progress :

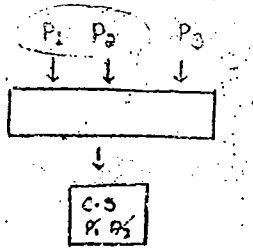
process running outside critical section should not block (the process of the other interested process from accessing / entering the critical section.





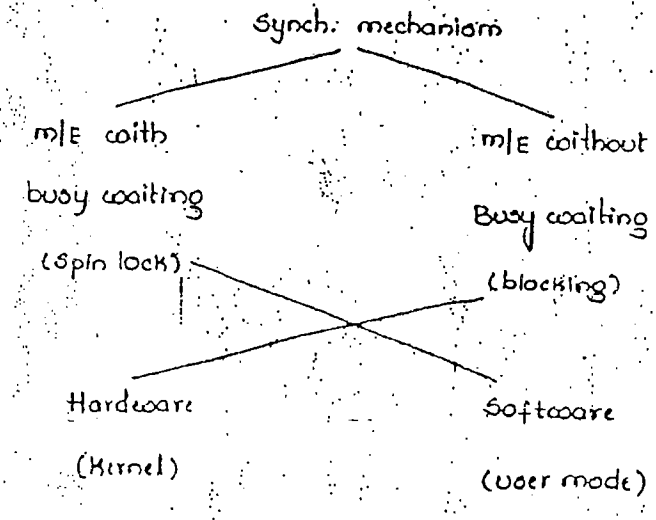
(3) Bounded waiting :-

* No process has to wait forever to access it's critical section.



If P_1 enters c.s and after leaving, if P_2 wants to enter c.s, then it can enter and after performing operations, it leaves. Then, if again P_1 wants to enter along with P_3, where P_1 has high priority than P_3, then P_1 enters & then P_2 and so on, where there is no chance for P_3 to enter c.s. (i.e., P_3 remains Idle forever).

(4) No Assumptions about the speed (or) no. of CPU's may be assumed.



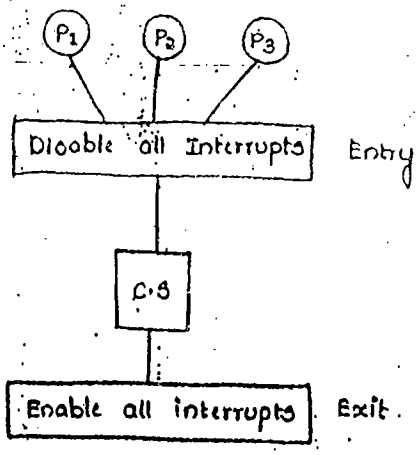
Disabling Interrupts :-

* m/E without busy waiting.
* multi-process
* Kernel mode.

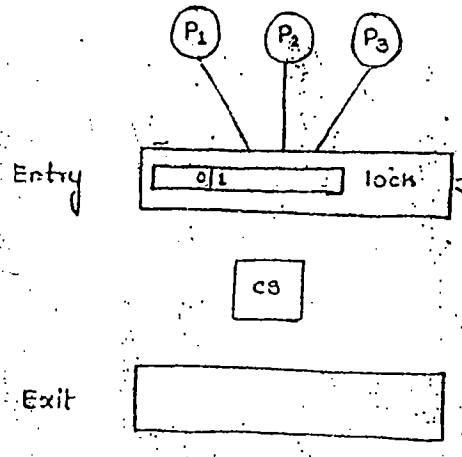
* The main cause of pre-emption is "interrupt".
* Disable the source of pre-emption in the critical section at entry level and enable the interrupts at exit level.
* It guarantees m/E because if P_1 running in c.s, other interrupts are disabled.
* Enabling / Disabling of Interrupts done only in Kernel mode.

So, o.s process can only enable / disable the interrupts and user process cannot enable / disable ⇒ Restriction to o.s process.

* critical section
* Race condition
* pre-emption.



* Slow mechanism implemented in user mode.
* Fix with busy waiting.
* multiprocess.



Doesn't perform direct manipulation on memory that load into ALU reg.

```
void Entry_section(int process)
{
    while(lock!=0); // Busy waiting
    lock=1;
}
cs
void Exit_section(int process)
{
    lock=0;
}
```

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

Step 1 : The lock value is stored in the register 'Ri'.

Step 2 : It is compared whether lock is '0' or not.

Step 3 : If lock value is not zero (i.e., lock=1), then again process same steps (1&2).

Step 4 : If lock=0, then the process enters c.s and while, entered then it changes lock=1.

Assembly language :-

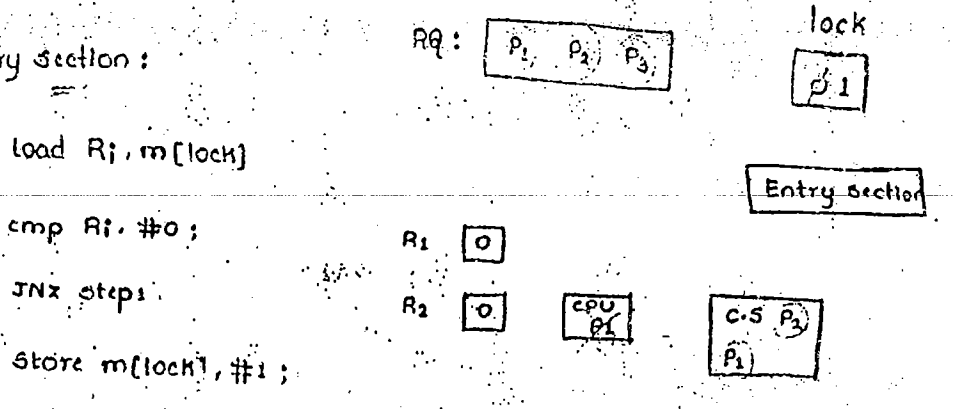
Entry section :

1. load Ri, m[lock]
2. cmp Ri, #0 ;
3. JNz step1 ;
4. Store m[lock], #1 ;

c.s

6. Exit section :

Store m[lock], #0 ;



* It fails to guarantee mutual exclusion.
* Busy waiting results in wastage of CPU cycles / CPU time.

Hint for testing either supports to guarantee m/E :-

(1) Identify the shared variable (lock)

(2) Read the value of shared variable (load)

(3) compare (optional)

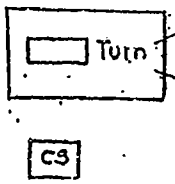
(4) Update / action (based on result of comparision)

⇒ If process 'P1' entered c.s, and before changing lock=1, it get pre-empted. Since the lock=0, another process 'P2' wants to enter c.s and get resides, in CPU, and gets into c.s.

* Since P1 and P2 gets into c.s simultaneously, it fails to guarantee mutual exclusion.

(3) Strict Alternation :

* Slow at user mode.
* 2-process
* mIE with busy waiting



0 process P_0's turn to enter c.s

1 process P_1's turn to enter c.s.

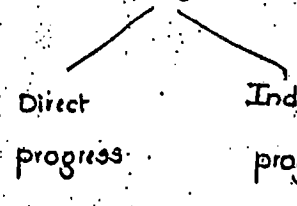
* Strictly an alternate basis, processes (P_0 & P_1) takes turn to enter c.s.

```
void P0(void)
{
    while(1)
    {
        Non_cs()
Entry   while(turn != 0); // busy waiting
        cs
Exit    turn = 1;
    }
}
```

Because "turn" value is not immediately updated, but updated at exit section, so, it guarantees mIE.

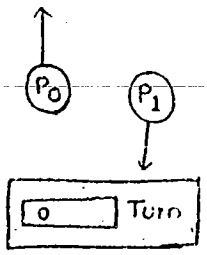
```
Void P1(void)
{
    while(1)
    {
        Non_cs()
        while(turn != 1);
        cs
        turn = 0;
    }
}
```

* Guarantees mIE.
* Not Guarantees progress



rect progress :-

Initially, the turn=0, and if P_0 is not interested in c.s, and P_1 is interested, then the turn must be equal to '1' to enter into c.s. But, unless 'P_0' enters, "turn" can't be changed, and 'P_0' not interested to do so. Hence, progress is not guaranteed.

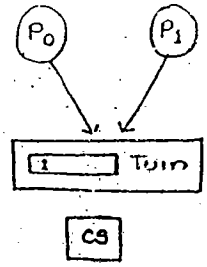


'P_0' which is outside block

'P_1' directly not to enter c.s

Indirect progress :-

If 'P_0' enters c.s and while exiting turn changed to '1', so that 'P_1' need to enter. If again 'P_0' wants to enter (i.e., interested) and P_1 is not interested. Then, P_0 is discarded to enter c.s because turn=1 and it needs P_1 to change turn=0. Hence, progress is not guaranteed.



P_1 is blocking P_0 indirectly.

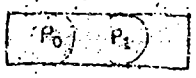
(4) Peterson's solution (Dekker's Algorithm):-

* m/E with busy waiting
* s/w at user mode.
* 2-process solutions (P_0 & P_1)

⇓

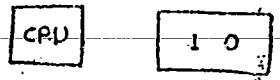
disadvantage.

```
#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 waiting
}
void Exit_section(int process)
{
    interested[process] = false;
}
```



int'd[0] = ✗ T

int'd[0] = ✗ T



P0 : 1, 2, 3

P1 : 1, 2, 3, 4, 5 ↻

P0 : 4, 5 ↻

P1 : CS

* Progress is guaranteed, mIE guaranteed.

* Bounded waiting is guaranteed (since, it is not guaranteed for more than 2 processes).

Drawback :

* wastage of CPU time.

(5) Test & Set lock instruction (TSL) :-

(privileged instruction ⇒ atomically)

* slow at user mode
* m/E with busy waiting
* multi-process (advantage over peterson)

Eg :- TSL register, flag ; ← 0 — cs is free
1 — cs is busy

** "copies the value of flag into register and sets the value of flag to one (always)".

Entry-section :

(1) TSL regi, m[flag];
(2) cmp regi, #0;
(3) JNZ step1
(4) CS
(5) store m[flag], #0;

Priority Inversion (P/TSL) :

* Preemptive priority based cpu scheduling is used.

If H>L ⇒ Deadlock

when a high priority process enters the CPU. And P_L requires CPU to perform it's instructions, inorder P_H to enter into c.s. so, P_H enters in CPU and needs to enter c.s & P_L is in c.s and needs CPU to exit. Hence, Deadlock occurs.

RQ :

P_H P_L

CPU
P_H P_L

P/TSL Entry Section

CS
P_L

Producer & Consumer :

* MIE without Busy waiting (Blocking)

(i) sleep() and wakeup() :-

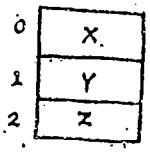
```
void producer(void)
{
    int itemp, in = 0;
    while(1)
    {
        produce item(itemp);
        if (count == N) sleep();  ⇒ If Buffer = full
        Buffer[in] = itemp;          producer ⇒ sleep()
        in (in+1) mod N;
        count = count + 1;
        if (count == 1) wakeup(consumer); ⇒ when item is placed in
    }                                      Buffer consumer ⇒ wakeup
}

void consumer(void)
{
    int itemc, out = 0;
    while(1)
    {
        if(count==0) sleep();
        itemc = Buffer[out];
        out = (out+1) mod N;
        count = count-1;
        if (count = N-1) wakeup(producer);
    }
}
```

RQ Ⓟ Ⓒ N=3

Ø 1 2 3

CPU



Buffer(0-2)

Ⓒ : ——— Preemptio
sleep()

Ⓟ : x
Inconsistency y
z
sleep(); } Deadlock
Ⓒ : sleep();

* while consumer executing instructions, before "sleep()" gets executed, pre-emption takes place and producer executes instructions and after buffer is full, producer goes to sleep() state assuming that consumer could wake-up, when consumer starts executing, sleep() instruction gets executed so, consumer expects that producer could be wake up. Hence, it represent the situation of "Deadlock."

17/07/2010

Saturday

Semaphores

* A variable (semaphore) that takes on integrated values.
  ⇓
  (Integers)

Semaphore

- Binary (0 & 1) ⇓ Critical section problem (mutex) → mutual exclusion.
- Counting (General Semaphore) (−∞ to +∞)

* It is an operating system resource.
* Semaphore operations executed in kernal mode (automatically)
  ⇓
  (no preemption)
* Implementation by Dijkstra (variable)

operations

- Down ⟷ up
- P ⟷ V
- wait ⟷ signal (relax)

```
Struct SEMAPHORE
{
    int value;
    QueueType L;
};          // list of PCB's of those processes that get blocked while
            performing "down" operation unsuccessfully.
```

```
SEMAPHORE s;
s.value = 4;
.
.
DOWN(s);
<c.s>
```

"Semaphore" gets executed in kernel mode.

* If the value after decrementing is negative, then it is unsuccessful.

* If value is zero (or) positive then it is successful.

(+) value → Successful Down processes.

(-) value → Blocked processes.

Eg:- s.value = 4 Down.

4-1 = 3	1	4 successful processes available in stack
3-1 = 2	1	
2-1 = 1	1	
1-1 = 0	1	
0-1 = -1	-1	2 blocked processes (unsuccessful processes)
-1-1 = -2	-2	

[(+)ve value of counting semaphore always indicate no. of Down operations successful].

⇒ waiting for semaphore.

```
DOWN(SEMAPHORE s)
{
    s.value = s.value - 1;
    if(s.value < 0) // unsuccessful.
    {
        put this process (PCB) in
        s.L() & Block it (sleep());
    }
}
```

After Blocking the process, Just it moves off (i.e., no Down operations are performed)

```
UP(SEMAPHORE s)
{
    s.value = s.value + 1;
    if(s.value ≤ 0)
    {
        select a process from s.L()
        and wakeup();
    }
}
```

Wakeup process gets into c.s but not

If s = -2 perform "Down" operation

After P_1 = -2+1 = -1

P_2 = -1+1 = 0

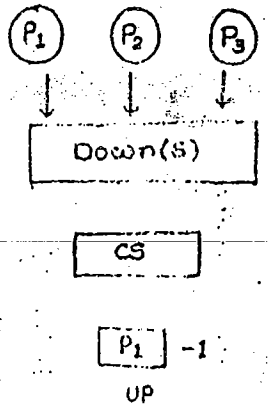
P_3 = 0

= 0+1

= 1

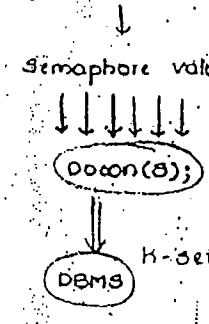
* When we have 3 processes, semaphore value starts with 1 ⇒ performing docon operation on it, the value becomes '0'. (i.e., the docon operation is successful, & it enters into critical section). After P_1 enters into c.s, all the other processes perform docon operation

($P_1 = 0$, $P_2 = -1$, $P_3 = -2$)



* Process never get blocked while performing "UP" operation and it gets blocked while performing "Docon" operation.

Allowing K processes into c.s



After process wakes up, don't perform any docon() operations, simply execute the critical section.

Eg:- S = 12.

16 p, 2 v, 8 p, 1 v, 4 p, 3 v, 8 p, 2 v

S.L() → Semaphore Queue consists list of processes.

S = 12 (16 docon) p → docon
= -4 (2 up) v → up.
= -2 (8 docon)
= -10
= -9
= -13
= -10
= -16 ⇒ -14

Binary (mutex) Semaphore (0/1):

```
struct BSEMAPHORE
{
    enum value (0,1);
    Queue Type L;
};
BSEMAPHORE S;
S.value = 1;
DOWN(S);
<c.s>
```

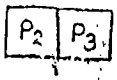
```
DOWN (BSEMAPHORE S)
{
    if (S.value == 1)  // Successful.
        S.value = 0;
    else  //unsuccessful.
    {
        put this process (PCB)
        in S.L() & Block it
        sleep();
    }
}
```

After performing down() operation on Bsemaphore,

if value = 0 ⇒ unsuccessful

value = 1 ⇒ successful.

If $S=1$, $\Rightarrow P_1 = 0$ (1-1) (After down() operation, semaphore must be 1).



P_2 checks semaphore value. Since, the value = 0, no down() operation is performed. Simply it moves to Block() state. Similarly, by having semaphore value, P_3 also gets blocked.

condition-I : If there are Blocked processes.

(or)

condition-II : If there is a process in critical section

Then, value of Binary semaphore must be zero.

After completion of all processes semaphore value gets incremented. (Because chance given to newly entered processes).

```
UP(BSEMAPHORE S)
{
    if(S.L() is not empty)
    {
        Select a process from S.L() and wake up()
    }
    else
    {
        S.value = 1;
    }
}
```

user doesn't know no. of processes present in the Queue by just having semaphore value. one can't decide no. of processes. So, we perform operation on blocked Queue

Semaphore value = 0 ⇒ It represents that there is no completion to present blocked processes.

After completion of processing all the jobs in blocked Queue, increment semaphore value. It represents that don't release semaphore until all the jobs are completed.

Initial semaphore = 0 no blocked processes

S = 1

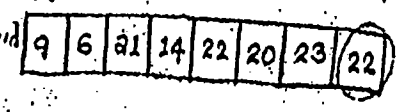
S=1 10p, 3v, 15p, 7v, 8p, 2v, 3p, 1v S=0

0 1 0 1 0 1 0 1

9 6 21 14 22 20 23 22

p = down

v = up



Cases of Binary Semaphore:-

Case (i) :-

S = 1	S value becomes = 0
P(S);	Status = successfully.

Case (ii) :

S = 1	S = 1
V(S);	Status = successful.

Case (iii):

S=0	S=0
P(S);	Status = unsuccessful.

Case (iv):

S=0	first value (initial value).	
V(S);	(Up operation)	S = 1 Status = successful.

Classical Interprocess Communication problems

(1) producer-consumer :-

```
#define N 100
Int Buffer[N];
Semaphore empty = N; // no. of empty slots in Buffer
Semaphore full = 0;  // no. of full slots
Semaphore mutex = 1; // used to ensure m/e between (P) & (C) on
                        buffer
void producer(void)
{
   int itemp, in = 0;
   while(1)
   {
      producer_item(itemp);
      down(empty);  // check for Buffer is full or not
      down(mutex)
      Buffer[in] = itemp;
      in = (in+1) mod n;
      up(mutex); // release the critical section.
      up(full);
   }
   consumer_item(itemp);
}
```

full = 0 ⇒ Buffer is empty
no item available

(Blocking producer)

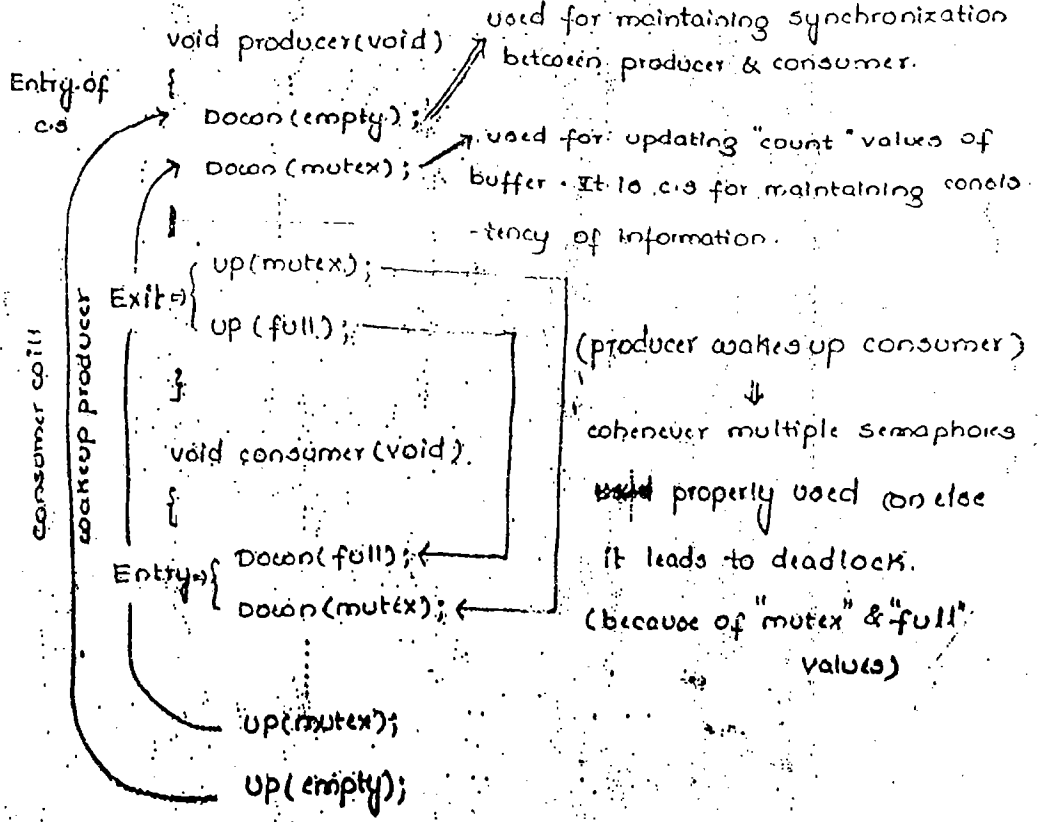
If Empty = 0 , down(empty) = 0-1 = 0

⇓

Buffer = full, then block producer.

Exit ⇐

```
void consumer(void)
{
    int itemc, out=0;
    cohile(1)
    {
        consumer_item(itemc);
Enter { Down(full);
        Down(mutex);   } Interchange if deadlock occurs to up(full) up(mutex).
        itemc = Buffer[out];
        out = (out+1) mod N;
        UP(mutex);
        UP(empty);
        producer_item(itemc);
    }
}
```



(2) Reader - writer problem :-

(1) 1 Reader - 1 writer

(2) Multiple Readers - 1 writer

(3) Multiple Readers - Multiple writers

* We can perform multiple "Read" operations at a time.

Database enters into critical section, when there is a writer. so, at a time, only one write operation can be performed.

```
int RC=0;   // Readers count
Semaphore mutex = 1;  // Used for mutual exclusion b/w readers)
                      RC must be in C.S.
Semaphore db = 1 ;  // Used to ensure M/E between readers & writers
                      on DBMS
Void Reader(void)
{
   while(1)
   {
        Down(mutex);   // To increment readers count, check "mutex" value
        RC = RC+1 ;
        if(RC==1)      // If first reader enters into DBMS, It's responsibility
          DOWN(db);    is to check whether any writer present in C.S
          UP(mutex);   (or) not. using (RC==1) condition.
        CS <READ-DBMS>
        DOWN(mutex);
        RC = RC-1;
```

```
        if(RC==0)
            UP(db);  ⇒ release db
            UP(mutex); ⇒ release mutex
    }
}
```

Suppose db have writer, it's val becomes = 0. so, no reader enter in the first stmt.

```
void writer(void)
{
    while(1)
    {
        Down(db);
        <update_data>
        up(db);
    }
}
```

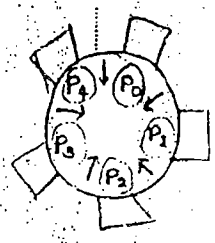
If first writer is successful to enter into c critical section, it release the mutex blw the readers for allowing many no. of readers

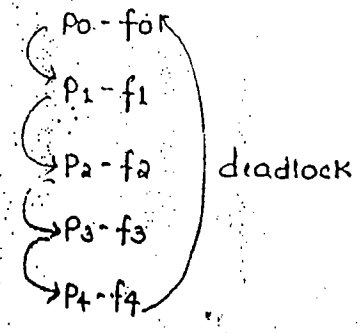
Again decrementing the 'Rc' count, use critical section of Readers.

(3) Dining philosopher's problem :-

Normal case :-

```
void philosopher(int i)
{
    while(1)
    {
        think(i);
        take_fork(i);
        Take_fork((i+1) % N);
        eat(i);
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```





- To avoid the problem of deadlock, in this, philosopher breaks the rule :-

  i.e., Even positions member follow one approach &
  odd position members follow another approach.

Actual problem :-

- Philosophers take the position on a table, they first try to take left hand side fork, if they succeed, then they take right fork. If both the forks are available, then they start eating & then put off the forks.
- But, at a time, all philosophers can't have both forks, where dead lock situation occurs.

Eg :- If P_0 first takes f_0, P_1 takes f_1, P_2 takes f_2, P_3 takes f_3, P_4 can't have it's left fork because P_0's right is P_4's left. Hence, dead lock occurs. So, no one wants to release their left forks.

```
State[N]      // stores state of philosopher either thinking
                (or) eating
Semaphore mutex = // used for releasing & taking the fork => it is C.S.
Semaphore S[N] = {0};
```

```
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (i+N-1) % N
#define RIGHT (i+1) % N
int state[N] = {0} ; // Either thinking (or) hungry (or) eating.
Semaphore mutex = 1 ; // To handle forks either by taking (or) releasing
void philosopher(int i)
{
    while(1)
    {
        Think(i); //initial state
        take_forks(i);
        eat();
        put_forks(i);
    }
}
void take_forks(int i)
{
    Docon(mutex); // taking forks.
    state[i] = HUNGRY ;
    test[i];
    UP(mutex);
    DOWN(s[i]); // fails to take forks => so blocked.
}
```

```
    DOWN(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    UP(mutex);
}

void test(int i)  // testing forks are available (or) not
{
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] ==
                                                        EATING)
    {
        state[i] = EATING
        UP(s[i]); // Eating is completed & set s[i] = 1
    }
}
```

State again changed to THINKING.

Concurrency & Concurrent programming

- Concurrency
  - Real
    - physical concurrency
    - (achieved through multiprocessor, array processor, vector processor)
  - Pseudo
    - concurrency (achieved by processor)

Precedence Graph dicates :

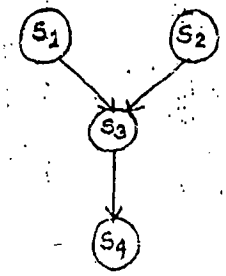
* It indicates which statements executes concurrently.

$S_1 : a = b + c;$

$S_2 : d = e + f;$

$S_3 : K = a + d;$

$S_4 : l = K * m;$



There is no precedence.

Any statement can be executed first.

Concurrency conditions :-

$S_1 : a = b + c;$

$S_2 : d = e * f;$

$S_3 : K = a + d;$

$S_4 : l = K * m;$

Read set → The values read into equation

$R(S_1) = \{b, c\}$.

write set → update value in equation

$w(S_1) = \{a\}$

$a + = \ldots\ldots b * c \ldots\ldots$ all a,b,c updated. So,

$R(S) = w(S) = \{a, b, c\}$

(1) $R(S_i) \cap W(S_j) = \phi$ If Reading & writing set of two different

(2) $W(S_i) \cap R(S_j) = \phi$ processes equal, then we can't perform

(3) $W(S_i) \cap R(S_j) = \phi$ (concurrent execution) at a time.

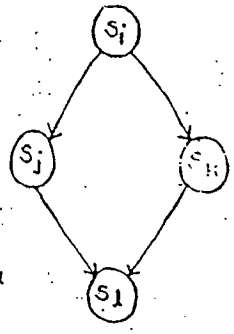
$a = c * d$

$a = l * m$ (not supported)

If (count $\neq 0$) ⇒ all processes are not completed.

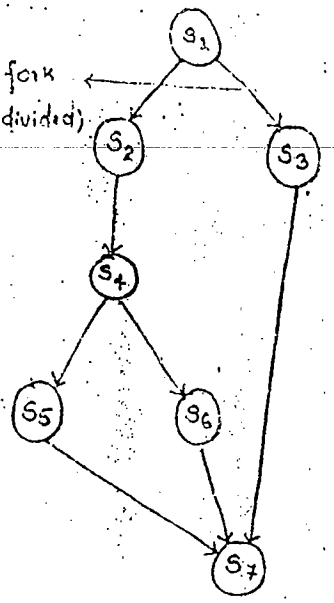
fork & Join :

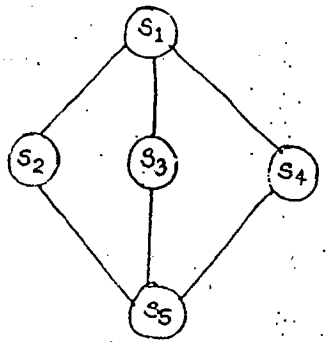
```
fork L1 ;
   Sj ; // after completion of Sj go to S1
go to L1
L1 : SK    // it executes immediate instructions Li
L1 : Join count ;  // in
     S1
```




```
Join (int count)
{
   count = count - 1 ;
   if (count ≠ 0)   // all the processes are not
   exit ;           | completed · so, there is
}                   | no execution of S1.
```

```
Count = 3
S1;
fork L1 (S1 followed by fork)
S2;
S4;
fork L2
S5;
L1 : S3
go to L3
L2 : S6
L3 : Join count
S7;
```






```
Count = 3
S1;
fork L1
S3;
go to L3;
L1 : fork L2
S2;
go to L2;
L2 : S4;
go to L3
L3 : Join count (check for all processes either complete (or not)
S5;
```

S_2, S_3, S_4 are Independent processes. So, they can be accessed in any way. But, S_5 is purely dependent on S_2, S_3 and S_4.

Eg:- If S_2, S_3 accessed & completed, and S_4 needed to be accessed now, even though there is S_5 to be accessed, we won't access. After the completion of all the processes at the upper level (S_2, S_3, S_4), then only the last level process (S_5) must be computed. For such checking, "Join"

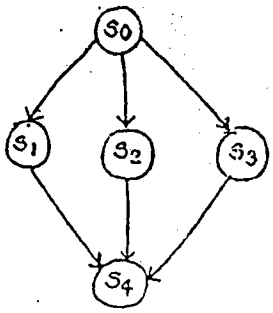
condition is used which counts the completed functions that are accessed, u
for completion of S_5.

Parbegin - Parend & co-begin - Coend :-

```
begin{}
  S1;
  S2;
  S3;
end}
```

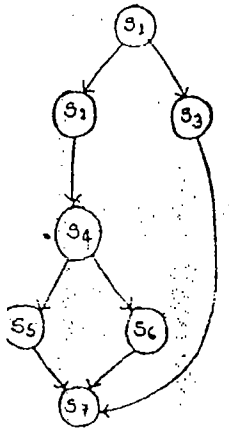

```
S0;
parbegin{}
  S1;
  S2;
  S3;
parend
  S4;
```

⇒ They support parallelism

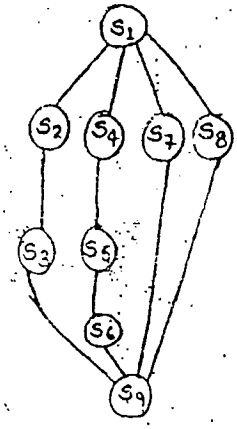


Begin-end ⇒ represents sequential actions

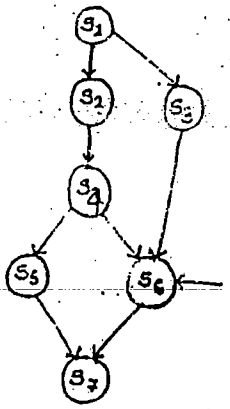
Parbegin - Parend ⇒ represents parallel actions.




```
S1;
Parbegin
  begin
    S2;
    S4;
    Parbegin
      S5;
      S6;
    Parend
  end
    S3;
Parend.
S7;
```



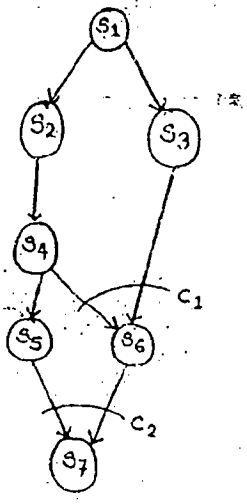

```
S1;
Parbegin
  begin
    S2;
    S3;
  end
  begin
    S4;
    S5;
    S6;
  end
  S7;
  S8;
Parend
  S9;
```



For this graph, we cannot write a code because for execution of 'S6', we need S4 and S3. With the help of code, we cannot impose such condition. We can overcome this problem by using "Semaphore".

Parbegin – Parend with semaphore :-

```
Semaphore a,b,c,d,e,f,g = {0};
Parbegin
   begin S1;
     Parbegin
        V(a);
        V(b);
     Parend
   end
   begin P(a); S2; S4;
     Parbegin
        V(c);
        V(d);
     Parend
   end
   begin P(b); S3;
        V(e);
   end
   begin P(c); S5;
        V(f);
   end
   begin P(d); P(e); S6; V(g); end, begin P(f); P(g); S7; end.
Parend                                    end
```



Implemention with fork & Join :-

```
C1 = 2; C2 = 2;
S1;
fork L1;
 S2;
 S4;
fork L2;
 S5
go to L3;
L1 : S3;
L2 : Join C1;
     S6;
L3 : Join C2;
     S7;
```

Eg:-

```
void P(void)
{
    A;
    B;
    C;
}

void Q(void)
{
    D;
    E;
}

main()
{
    Parbegin()
    P();
    Q();
    parend()
}
```

What are the possible outputs for the above code:

(a) A,B,C,D,E (b) D,E,A,B,C (c) ABCDE (d) D,A,C,E,B

(e) E,C,B,D,A

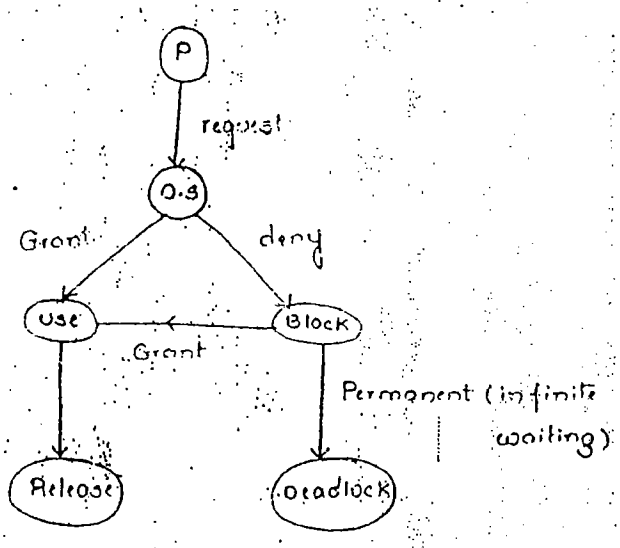
A,B,C and D,E are sequential actions. C,B cannot be happen. so, only first three are possible outputs.

DEADLOCK (Lockingup)

* Two (or) more processes are said to be involved in deadlock, iff they wai for the happening of an event, which would never happen.

* System model (for avoiding deadlocks)
  * 'n' processes ($P_1 \ldots\ldots P_n$)
  * 'm' resources

* Starvation ⇒ long waiting.
* Deadlock ⇒ Infinite waiting.



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

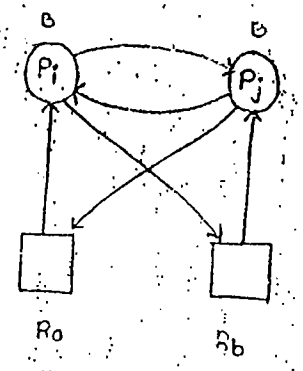
Necessary conditions for occuring deadlock :-

(1) Failure of mutual exclusion.

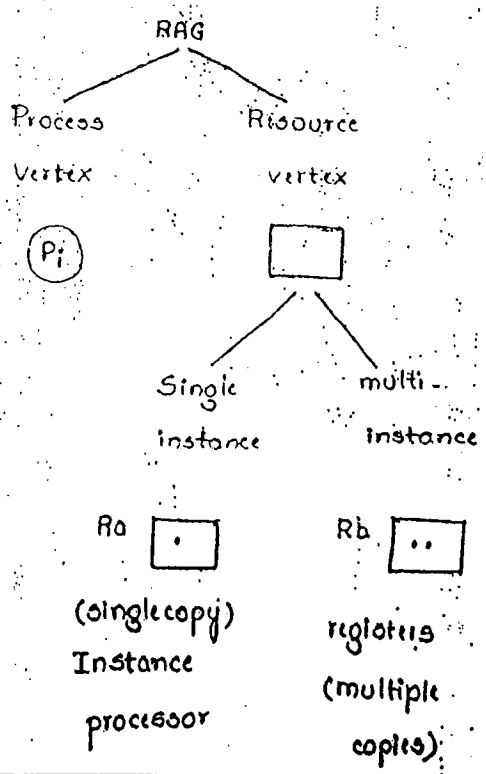
(2) Hold & wait.

(3) No-preemption (process won't release their resources).

(4) circular wait.

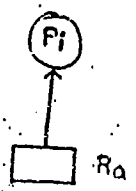


Resource Allocation Graph (RAG) / Multigraph :-

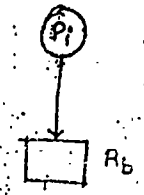


Edges :

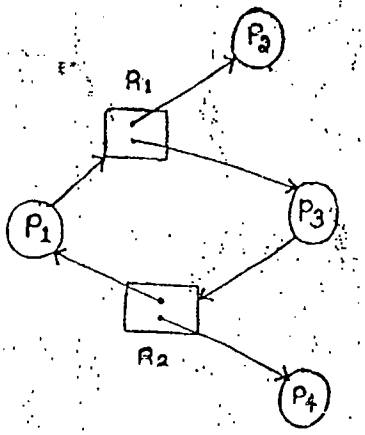
Assign edge

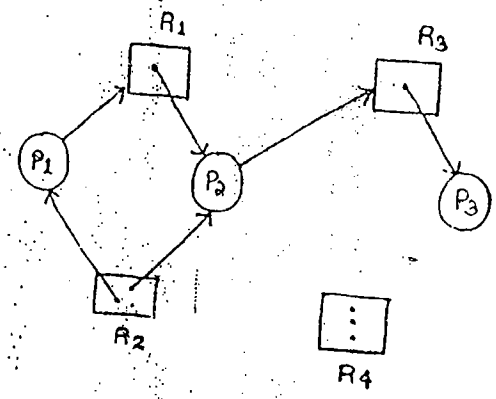


Request edge.



→ Pi allocating 2 instances of "Ra".





P_1 gets blocked, since it requests 'R_1', which inturn R_1 gets accessed by 'P_2'. Similarly, P_2 also gets blocked.

* If RAG has single instance and multiple instance.
* In multiple instances, there is a chance of occuring deadlock, because they form cycles (some times).
* Single instances may also cause deadlock, if the processes accessed in the cycle form.

Type-1	Type-2	Type-3
* Deadlock prevention * Deadlock avoidance (Banker's algorithm) No deadlock happens/ occurs.	* Deadlock Detection and Deadlock Recovery. Deadlock happens	* Deadlock Ignorance (Ostrich Algorithm) ↓ (Desserts)

Deadlocks prevention :-

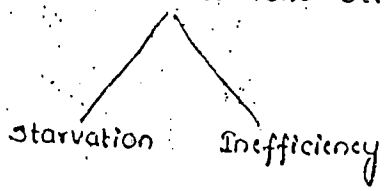
* Try to dissatisfy one/more of the ~~more~~ necessary conditions.

(1) mutual exclusion ✗

(2) ! (Hold & wait) :

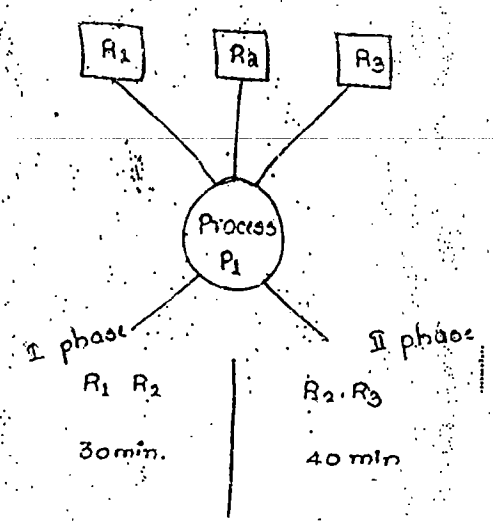
Either "hold" (or) "wait" (but not both)

* Requests & de-allocation of all the resources done between the process commencements.



mutual Exclusion :-

* Holding only one process within the critical section.



Consider,

* A process 'p1' which needs to access the resources 'R1', 'R2' and 'R3'. If a resource 'r1' and 'r2' are accessed and another time, again 'R2' needs to be accessed. so, the following situation holds :-

Hold & wait :-

* Process 'p1' in first phase, requires R_1 and R_2 resources and the time span allotted to phase-I is 30min to complete.

* After the completion of phase-I, process 'p1' releases the resource 'R1' but holds the resource 'R2' (because it again requires the same resource at phase-II also), and requests for the resource "R3". This situation is considered as "Hold & wait".

* Inorder to avoid this problem, we have two types of solutions :-

I^st Approach :-

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

Eg :- $P_1(R_1, R_2, R_3)$

* They doesn't wait for all the resources to be accessed.

II^nd Approach :-

* Release all the resources before making a new request.

Phase - I ⇒ $P_1(R_1, R_2)$

phase - II ⇒ $P_1(R_2, R_3)$

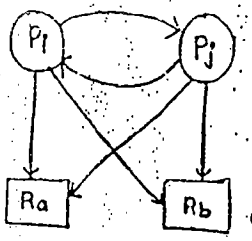
* After the completion of phase-I, release all the resources eventhough, it needs the same resource at phase-II.

* Then, again in phase-II, the resources are requested newly (if already used in phase-I).

(c) No pre-emption :-

* Allow the processes to get preempted

Self forcefully



(host)

Self release ⇒ It requests the resources that are not available, so that process releases other resources that are already available

Consider, P_i, which already accessed 'R_a', it requests for R_b, but R_b is not available, at that time, so, 'P_i' releases 'R_a' resource (in the sense, it may be useful for others) after completion of other process, it will execute.

22/07/2010

Thursday

Prevention (restrictive):

(d) circular wait :

"Total order relation on all process for requesting of resources."

Incl dec.

$f(R) \longrightarrow$ 'I'

$R_1 - 8$

$R_2 - 4$

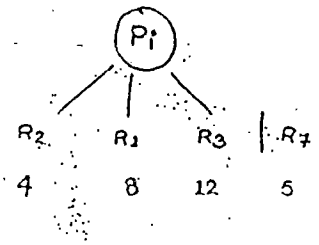
$R_3 - 12$

$R_4 - 18$

$R_5 - 25$

$R_6 - 9$

$R_7 - 5$



P_i requests in any order (i.e., either in incl decr).

If no order is followed.

any sequence is followed. P_i, must release all resources.

Starvation occurs when there is a release on "

Avoidance :-

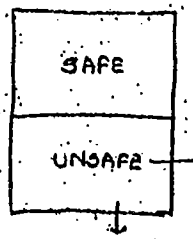
(Banker's Algorithm):

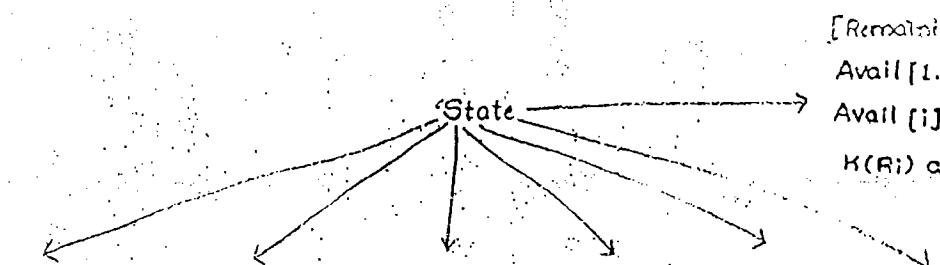
* Safety Algorithm
* Resource - Request

Objective :-

* Obtain a system always in a SAFE mode.

+ It is an "A priori Info" algorithm, It is based on state of the system.





'n' no. of processes

'm' no. of Resources (with Instances).

maximum demand of each process for a resource

Total (instances) of each resource

Allocation of resources to each other

Need [1...n, 1...m]

Need [i,j] = K

$P_i \longrightarrow K[R_j]$

Alloc [1:n, 1:m]

Alloc [i,j] = K

$P_i \longleftarrow K(R_j)$

Max [1..n, 1..m]$_{n \times m}$

Max [i,j] = k.

$P_i \xrightarrow[max]{} K(R_j)$

In Avoidance, each process must reveal the no. of resources required for it.

Eg: n = 5 (P_1 to P_5).

m = 1. R = 28 (Total) Need = max - Alloc.

n	R (max)	R (Alloc)	(Need) R	(Avail) R
P_1	10	5	5	3
P_2	20	10	10	4
P_3	3	1	2	8
P_4	8	4	4	13
P_5	12	5	7	18
				28

T_0 : $\langle P_2, P_4, P_5, P_1, P_3 \rangle$ safe-sequence

Avail = current avail + allocation.

* System is said to be SAFE, iff the needs of all the processes can be satisfied with the available resources in some (or) the other order (is called SAFE sequence) otherwise, the system is said to be

warning ⟸ But not really a deadlock — to indicate

n=5, m=3 & (A,B,C) = (10,5,7)

n	max	Alloc	need	Avail	Alloc
	A B C	A B C	A B C	A B C	(A+B+C)
					10 - 7 = 3
P_0	7 5 3	0 1 0	7 4 3	3 3 2	5 - 2 = 3
P_1	3 2 2	2 0 0	1 2 2	5 3 2	7 - 5 = 2.
P_2	9 0 2	3 0 2	6 0 0	7 4 3	
P_3	2 2 2	2 1 1	0 1 1	7 4 5	
P_4	4 3 3	0 0 2	4 3 1	7 5 5	
				10 5 7	

T_0 : $\langle P_1, P_3, P_4, P_0, P_2 \rangle \Rightarrow$ safe.

Initial availability = (3,3,2), o.s have enough resources to grant to P_1. bc it first calculates any problem if user gives this resource. It runs the "safety algorithm" in this situation.

T_1 : Resource Request algorithm :-

"Request by the resources are granted or not" is being identified i this algorithm.

T_1 : P_1 : $\longrightarrow$ (1,0,2) (Req 1) $\longrightarrow$ $\langle P_1, P_3, P_4, P_0, P_2 \rangle \Rightarrow$ safe.

T_1 : P_4 : $\longrightarrow$ (3,3,0) (Req 4) $\rightarrow$ Blocked.

It is blocked because it doesn't satisfy the Availability.

T_1 : $\langle 0,2,0 \rangle \longrightarrow P_0$

	max	Alloc	need	Avail
	A B C	A B C	A B C	A B C
P_0	7 6 3	0 1 0	7 4 3	2 3 0
P_1	3 2 2	3 0 2	0 2 0	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

Resource Request Algorithm :-

1. Request ≤ Available
2. Request ≤ need — $P_1 \rightarrow \langle 1,0,2 \rangle$ request
3. (Assume request is satisfied) — $P_4 \rightarrow \langle 3,3,0 \rangle$

   Available = Available - request — $P_0 \rightarrow \langle 0,2,0 \rangle$

   Allocation = Allocation + request

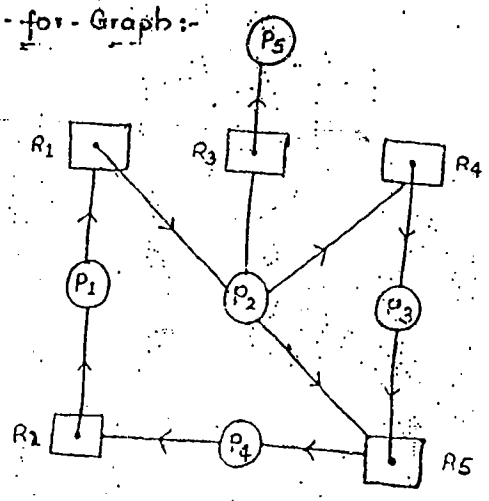
   Need = Need - Request
4. If result is safe, allocate resource.

III. Deadlock Detection & Recovery :

occurs { when system performance is low.
If many blocked processes are present.

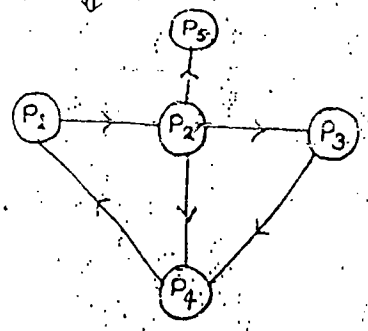
(1) Single Instance of a resource :

wait-for-Graph :-



Among 5 processes, 4 are blocked. So, we apply deadl detection algorithm.

Cycle Detection Algorithm is used ⇒



$P_1 - P_2 - P_3 - P_4 - P_1$

⇓

Cycle ⇒ Deadlock

II multi-Instance of a resource :-

n = 5, m = 3, (A,B,C) = (7,2,6).

⇒ request is again processed after allocation

T_0	Alloc	Req	Avail	
	A B C	A B C	A B C	7-7=0
				2-2=0
P_0	0 1 0	0 0 0	0 0 0	6-6=0
P_1	2 0 0	2 0 2	0 1 0 ✓	
P_2	3 0 3	0 0 0	3 1 3 ✓	
P_3	2 1 1	1 0 0	5 2 4 ✓	
P_4	0 0 2	0 0 2	5 2 6 ✓	
			7 2 6 ✓	

Initially, resource availability is '000' but P_2 requests 001, It can't satisfy O.S.

$P_1 \rightarrow P_4$ ⇒ Input's to recovery module.

To check safe mode : $\langle P_0, P_2, P_3, P_4, P_1 \rangle$

T_5 : $P_2 \longrightarrow \langle 0, 0, 1 \rangle$

$\langle P_0, \ldots\ldots$ × Deadlock

$P_1 - P_4$

⇓

recovery

Recovery

Process Termination

Kill-All

Kill-one-at-a-time

↓

repeatedly run the algorithm

Resource pre-emption

with

Roll back

(disadv : starvation)

Kill-All :-

Adv :-

* No need for detection again.

Disadvantage :-

* Again, it must be started from beginning.

* To overcome, this, we have "killing gracifully".

"One-at-a-time":

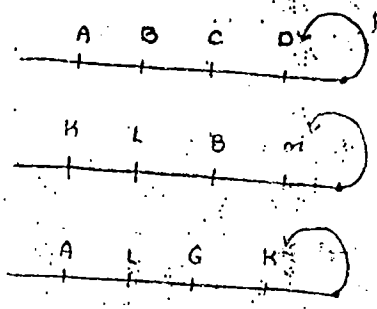
Adv:-

* Save the other processes to resume.

Disadv:-

* Need for detection algorithm as

Resource pre-emption:-



Repeatedly, a process is detec
to have deadlock and re-star
then the starvation occurs

Deadlock handling strategies are not used on desktop systems and "Deadlock Ignorance" is considered :-

* Robustness.
* follows ostrich algorithm (in missile, real time, Industrious o.s) ⇒ not important tasks

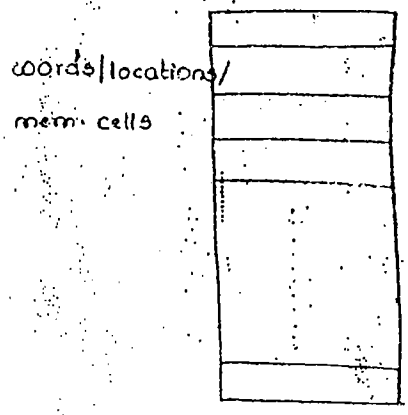
The strategies are used in real time systems where there is no single bit errors

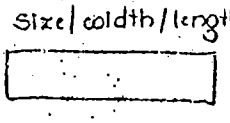
⇓ follows Deadlock prevention

Memory Management

Abstract view of memory

(1-D Array of words)



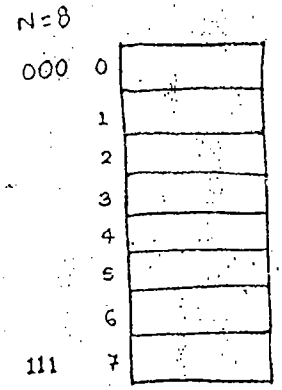


(Fixed length binary bits)

N = Total no. of words

n = Address (in bits)

m = width/size (bits) of words



N = 64 words

↓

n = 6 bits (2^6)

N = 512 M words

↓ ↓

$= 2^9 + 20$

= 29 bits

N = 4 GW

↘ ↘

$n = 2^2 + 30$

= 32 bits ⇒ 2^{32} = 4GW

$2^5 = 32$ W

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

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

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

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

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

$N = 2^n$

$512 \Rightarrow 2^9$

$n = \lceil \log_2 N \rceil$

memory specification

coord Byte Bit

I

$n = 23$ bits.

$m = 16$ bits (1 coord = 2 bytes)

	capacity	Address	
words	8MW	$n = 23$ bits	$8 = 2^3$ MW → $3 + 20 \Rightarrow 23$
Bytes	8M × 2B = 16MB	$n = 24$ bits	$16 = 2^4$ MB → $4 + 20 = 24$
Bits	16M × 8 bits = 128 Mbits	$n = 27$ bits	$128 = 2^7$ mbits → $7 + 20 = 27$

II $N = 256$ MW

$m = 32$ bits (1 coord = 4 bytes) $\Rightarrow$ 256 M × 4B $\Rightarrow$ 1G.

	CAPacity	Address	
coords	256 MW	$n = 28$ bits	32 bits
Bytes	256 M × 4B = 1G.	$n = 30$ bits	2^5.
bits	8G bits	$n = 33$ bits	

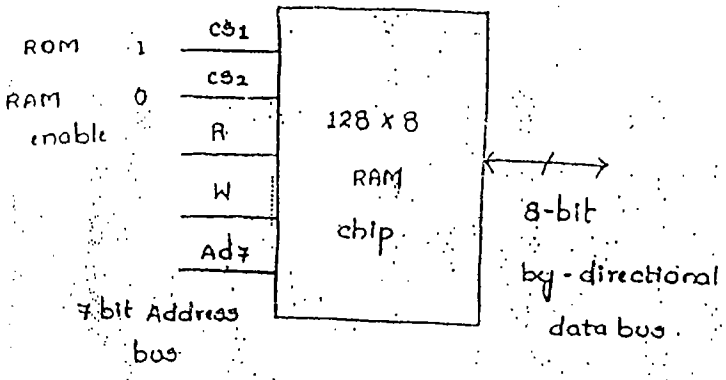
N = 256 G bits

m = 64 bits (1 word = 8 bytes) => 256 G x 8 B

$= \frac{2^8 \cdot 2^{30} \times 2^3 \times 2^3}{2^6} = 2^{32} = 4\,GW$

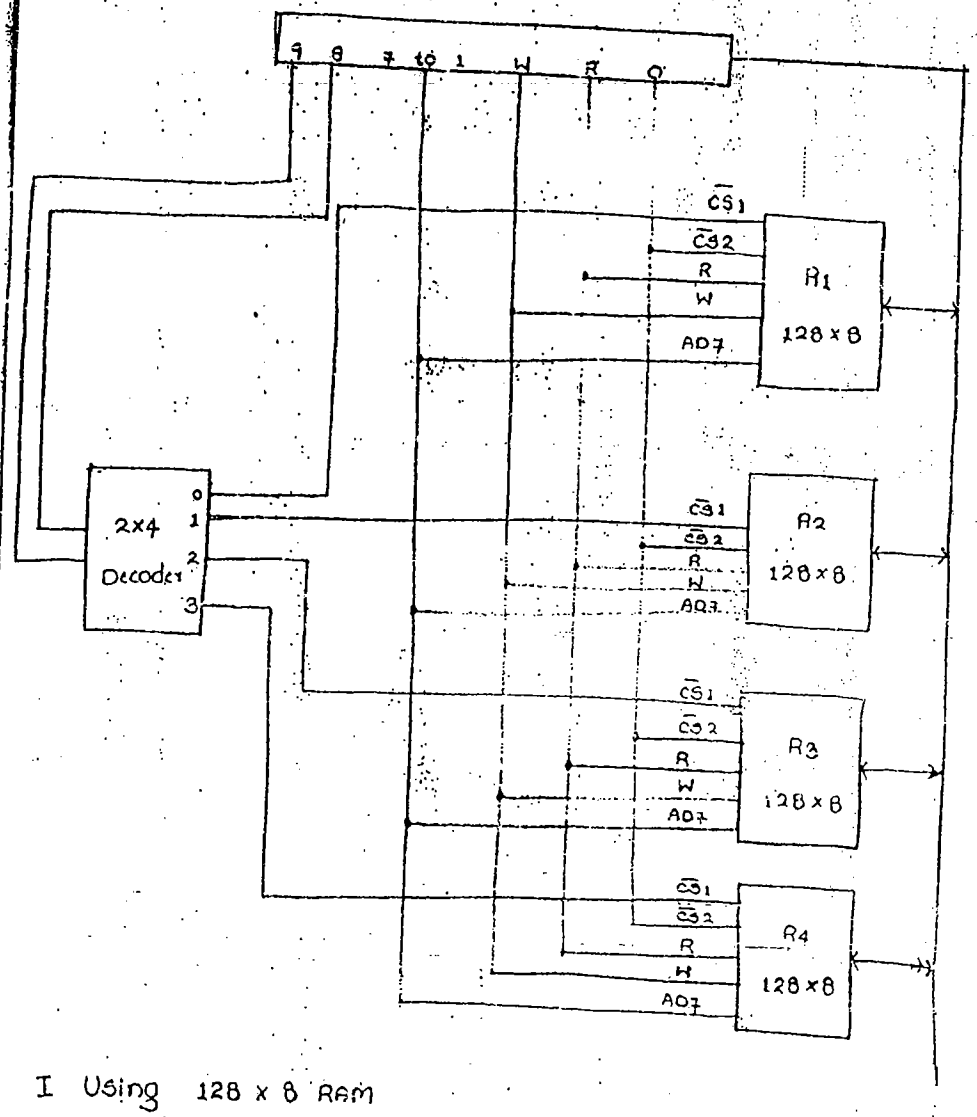
	capacity	Address
words	4GW	32 bits
Bytes	4G x 8B = 32 GB	35 bits
Bits	256 Gbits	38 bits

RAM chip organisation :-



No. of chips = $\frac{\text{memory desired}}{\text{chip size}}$

$= \frac{512\ MB}{128\ MB} = \frac{2^9}{2^7} = 2^2 = 4$



I Using 128 x 8 RAM

16 KB = ?

(i) chips = 128

(ii) n = 14 bits

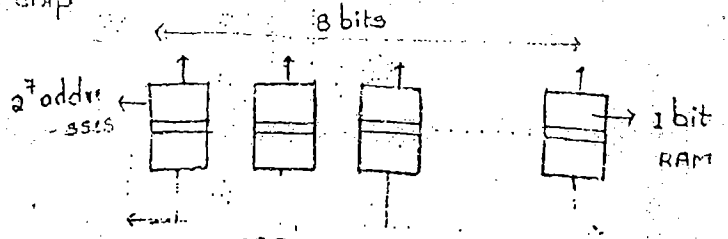
(iii) Decoder = 7 x 128

↓ no. of words increasing

$$\frac{16\ KB}{128} = \frac{2^{14}}{2^7} = 2^7 = 128$$

i) Using 128 x 1 RAM chip

If contains each word = 1 bit



128 bytes

(i) chips = 8

(ii) n = 7 bits

128 B

$\frac{128 \times 8}{128} = 1 \times 8 = 8$

since, memory and data size are same, there is no need of decoder.

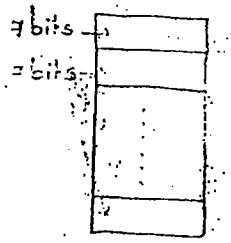
ii) Using 128 x 1 RAM chip

16 KB

(i) chips = 1024

(ii) n = 14 bits

(iii) Decoder = 7 x 128



128 x 8

$\frac{16\ KB}{128 \times 1}$

$= \frac{2^{14}}{2^7} = 2^7$

$= 128 \times 8$ bits

128 x 1 RAM

128 * 8 rows = 1024 bits

$\frac{16KB}{128 \times 1} = \frac{2^4 \times 2^{10}}{2^7} = 1024.$

$= 2^7 = 128 \times 8$

$= 1024$

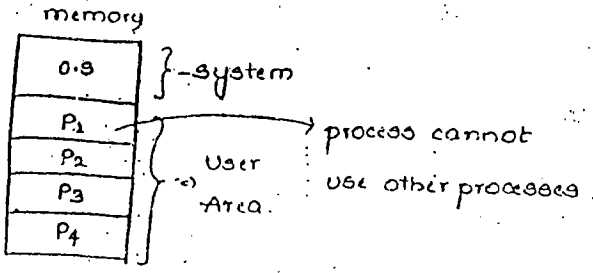
Memory Manager

Functions :

* Allocation
* Protection
* Free space management
* Deallocation

Goals :

* Efficient utilization of space.
* Run larger program in smaller memor



Goals :-

(1) Space utilization :-

Keeping fragmentation at each level, so that there is a chance of effective utilization of space.

fragmentation → memory loss can occur.

Internal External

(2) Run larger program in smaller memory :

Program = 100 KB (blok)

⇩

using virtual memory / overlays.

Eg. :- Run the data within pendrive (8 GB - 60 KB)

Memory management

Techniques

Contiguous (CS)	Non-contiguous (NCS)
* centralized	* Decentralised
program, as a unit is completely stored	program portions are distributed to memory.
* Overlays	* paging
* partitioning	* segmentation
	* segmentation-paging
	* virtual memory.

24/07/2010

Saturday

(1) Overlays (larger programs run in smaller space):-

2 pass Assembler

pass 1 : size - 70 KB

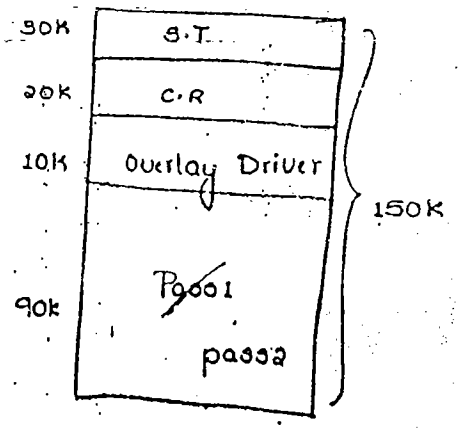
pass 2 : size - 80 KB

Symbol Table

for construction of passes : 30KB

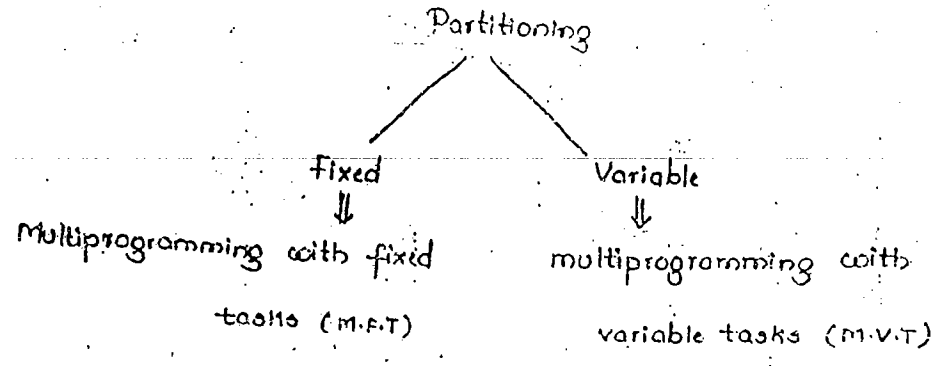
C.R (Common Rupees) : 20KB

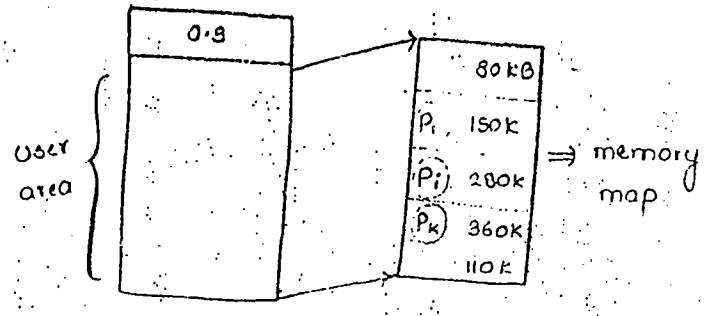
200 KB



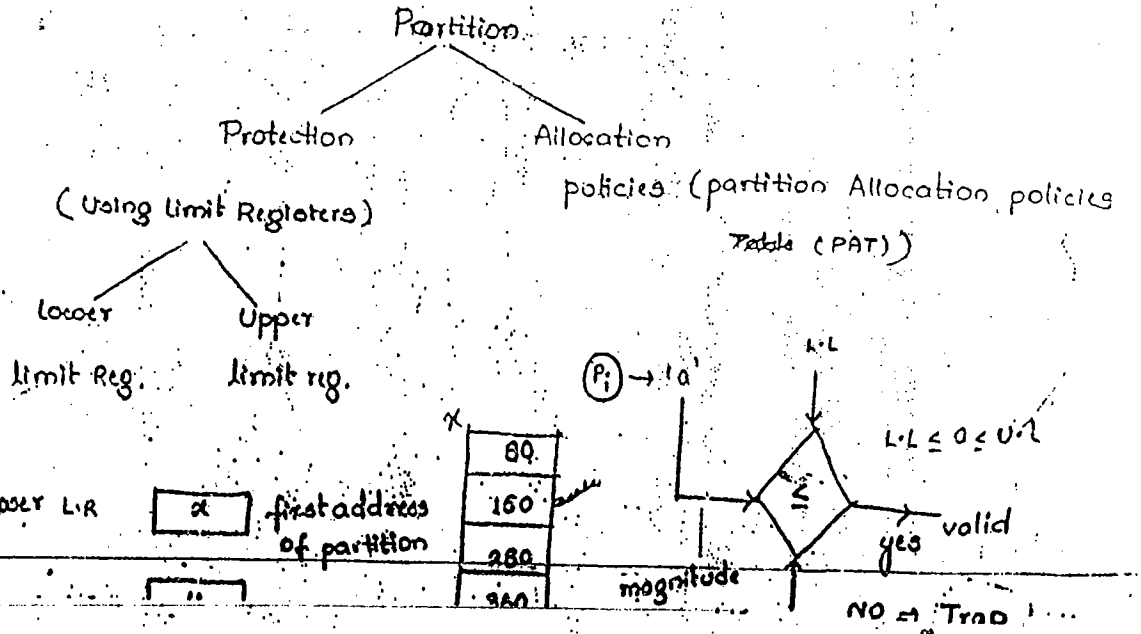
loading programs → overlay driver

(a) partitioning :-

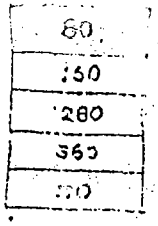




Fixed (M.F.T) ⇒ no. of partitions are fixed and of different sizes since 1 partition = 1 process



Eg: $P_{next} = 100$ KB



Partition Allocation policies :-

(i) First fit (F.F) :-

* Allocate the process in the first free begin of the partition.

To include $P_{next} = 100$KB, the partition of P = 150 KB is selected as it is the first partition of required size.

ii) Best fit (B.F):

* Smallest begin of free partition.
    * Best available fit
    * Best fit only.

Best available fit :-

Among available partitions, select the smallest best available.

Best fit only:-

It selects for best fit only, if it is not available, it waits for that partition, and after available it takes.

) Next fit (N.F):

Same as first bit, but it doesn't start from first partition, but it starts from the place where the last partition ended up (i.e., last process's partition ended point).

Advantage :-

* Searching becomes faster than first fit.

(iv) worst fit (w.f):

Largest free partition ⇒ selecting the largest available free partition and get placed in that partition.

Performance of MFT :-

(1) fragmentation :-

* Internal
* External ⇒ when unable to accomodate a process of size even though we have sufficient size of memory.

1 partition ⇒ 1 process

(contiguous allocation)

	80 KB
	150 KB
*	280 K
*	360 K
	110 KB

80 + 150 + 110 = 340 KB

Eg:- Process = 180 KB

If we have a new process of 180 KB to place into memory, but there is no certain partition to get fit into this process even though it has total of 340 KB free. So, it is called as External frag.

(2) Maximum process size (or) maximum partition size ⇒ no flexibility.

(3) Degree of multiprogramming :-

Restricted to no. of partitions (can't be > n).

⇓

no flexibility.

(4) Best fit is superior to all.

Inorder to overcome all these problems, we have M.V.T.

MVT (variable partitions):-

⇒ Dynamic approach

⇒ No user area for partitioning

P_1	80 K
P_2	110 K
P_3	75 K
P_4	180 K
P_5	260 K
P_6	330 K
300 K	

PCM (Partition Control memory)

Used for storing the control information of particular partition.

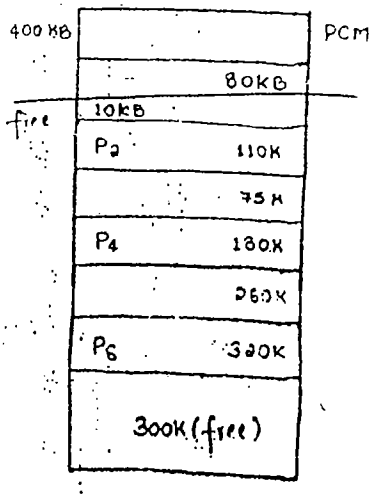
* No use of limit registers in MVT for protection.

* limit value
  size
  Status
  L.L, U.L

  } stored in PCM

Eg:- If a new process of 70KB is needed to fit into memory, then if we take first fit. the 80KB partition is divided as 70KB for the new process & rest of 10KB : so, we overcome the fragmentation problem here.

Process (new) = 70 KB



Performance of MVT :-

(1) fragmentation:-

* External
* No Internal.

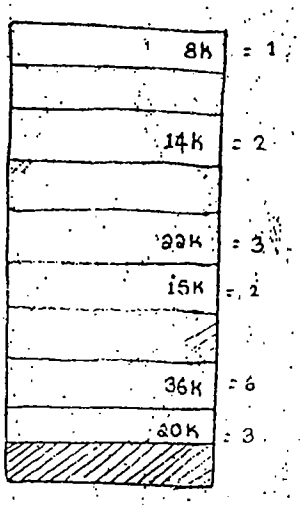
(2) Max. process size ⇒ flexible

no max. part. size.

(3) Degree of multiprogramming :-

* No restriction to no. of partitions.
* flexible

(4) worst fit performs better, and best fit performs worst.



(i) 6K ?

First fit = 8K

Best fit = 8K

next fit = 36K

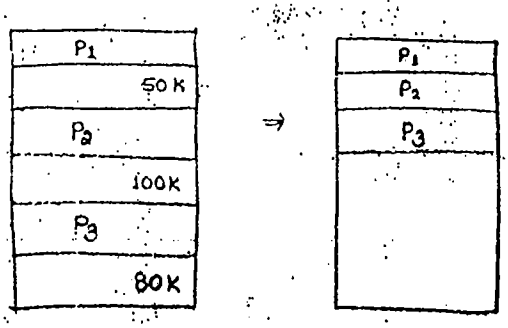
worst fit = 36K

(ii) How many successive requests of size 6K can be satisfied.

Ans: 17.

External fragmentation:-

(i) Compaction (dynamic relocation of processes):-



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

* process should be dynamically relocatable and can be processed / continued.

If cannot be run (or processed, then no. compaction is followed.

Disadvantages :-

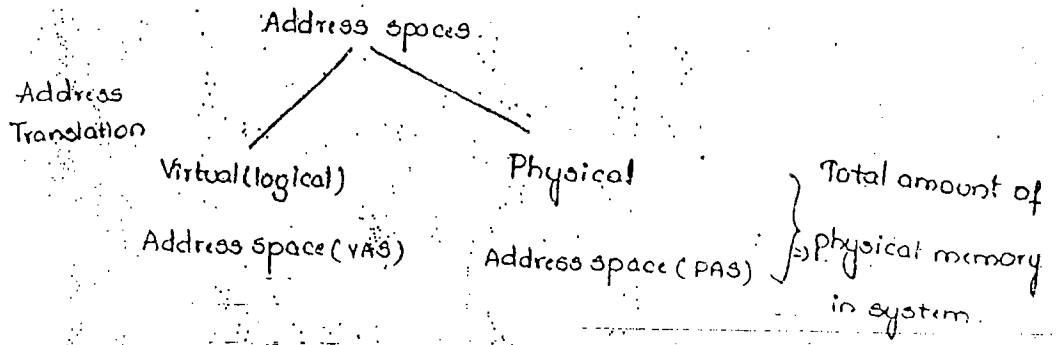
* longer time (Since everytime relocation is done).
* If done frequently, time is consumed.

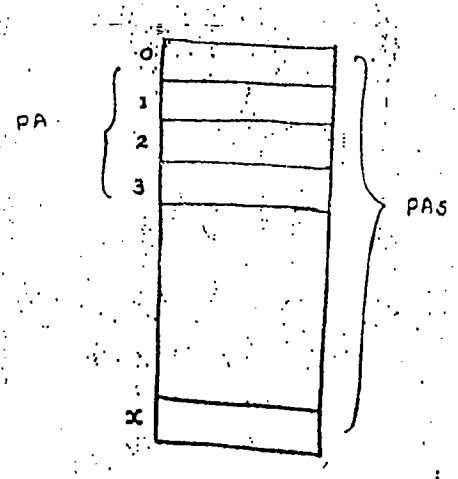
(2) Non-contiguous Allocation :-

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

Address space :-

* space of coords / series of coords
* A set (set) of locations of memory is address space.





$PAS \times PA \Rightarrow PAS = a^{PA} \Rightarrow PA = \log_a [PAS]$.

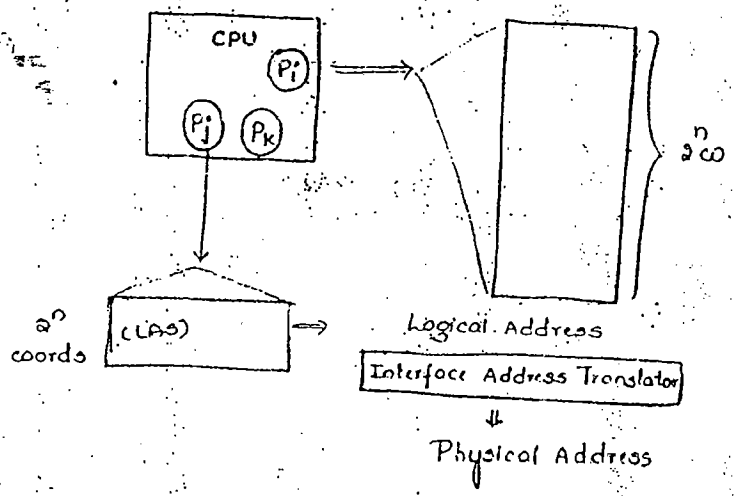
Logical Address Space :-

* Every process within the CPU has an assumption that it can access the whole memory available.

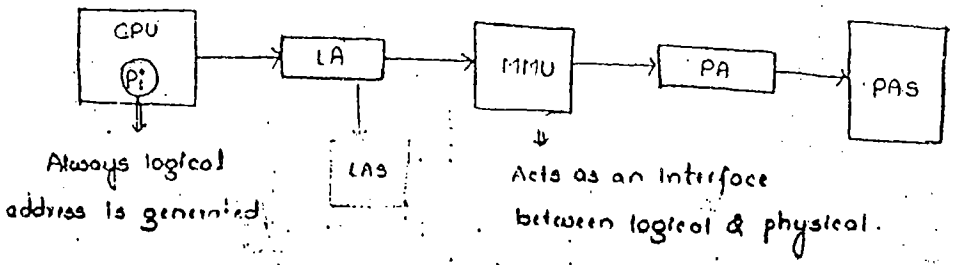
* LAS must be translated to physical for instruction/data fetch.

If LA = 13 bits (2^3K) $\quad LA = \log_2(LAS)$

Then, LAS = 8 KW $\quad LAS = 2^{LA}$



Architecture of non-contiguous Allocation policies :-



(1) Organisation of LAS

(2) Organisation of PAS

(3) Organisation of MMU (mem mgmt unit)

(4) Translation process

LAS (or) VAS	>	PAS

(practically)

Because, it must have to support the concept of larger program storage in shorter memory.

Theoritically, all of them have different relation.

Paging:-

Simple paging:-

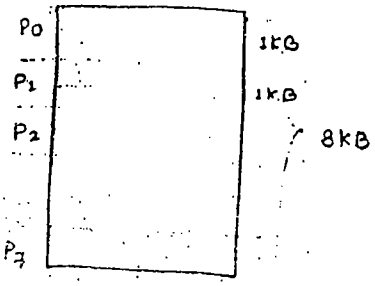
LAS = 8 KB , PAS = 4 KB ; LA = 13 bits PA = 12 bits

(1) organisation of LAS:-

(i) LAS is divided into equal size pages

(ii) Page size is a power of 2.

(iii) No. of pages (N) = $\frac{\text{LAS}}{\text{Page size}}$



(iv) page no. (p) = $[\log_2 N]$

$N = 2^P$

(v) page offset (d bits) = $[\log_2 PS]$

$PS = 2^d$

(vi) Format of logical address:

LA = 13 bits

P	d
3	10

words

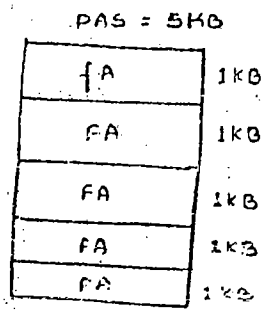
pages.

(2) Organisation of PAS:

(i) PAS is divided into equal size frames (page frames)

(ii) Frame size = page size

frame size is same as page size



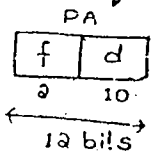
(iii) No. of frames (m) = $\frac{PAS}{FS}$

(iv) frame no. ('f' bits) = $\lceil \log_2 N \rceil$

$N = 2^f$

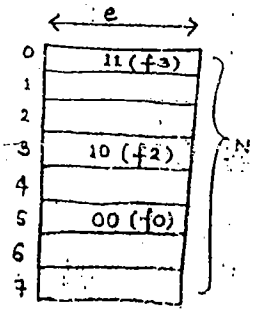
(v) Frame offset = page offset

(vi) Frame holds the page, therefore, frame offset is equal to page offset.



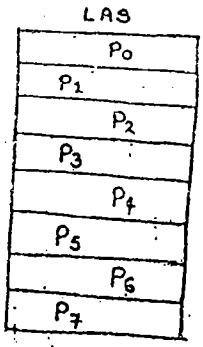
(3) Organisation of MMU (page table):-

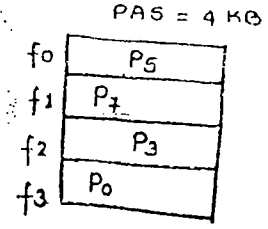
page map Table => Associated with processes.



$e \times N$

$2 \times 8 = 16$



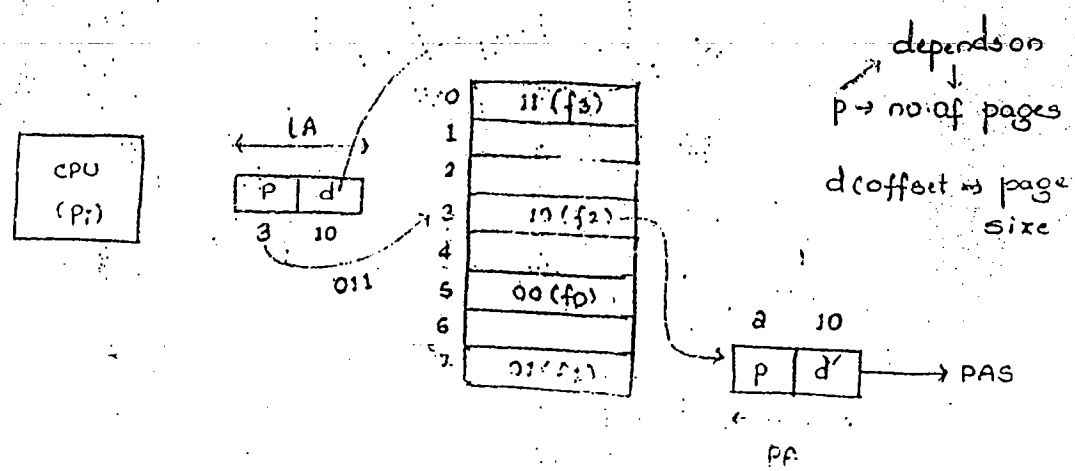


* No. of entries in P.T (P.T size) = No. of pages in LAS.
* PT entries contain frame numbers.
* Associated with processes (Each process has it's own page table)
* page tables are stored in main memory.

* P.T size (in bytes) = N * e bytes

e → page table entry size in bytes

Address Translation:-



Eg1 :-

LAS = 32 MB, PAS = 256 KB N = ? m = ? p = ? f = ? d = ?

page size = 4 KB P.T size = ?

Sol:-

$$N = \text{no. of pages} = \frac{32\,MB}{4\,KB} = \frac{2^{25}}{2^{12}} = 2^{13} = 8K.$$

$$M = \frac{256\,KB}{4\,KB} = \frac{2^{18}}{2^{12}} = 2^{6} = 64$$

p = page no. (depends on no. of pages) = 13 bits ($N = 8K = 2^{3+10} = 13$)

f = frame no. (depends on m) = 6 bits.

d (offset) depends on page size = 12 bits ($4\,KB = 2^{2+10} = 12$)

P.T size = no. of pages = 8K

If e = 4B then P.T Size = e * N

= 8K * 4B

= 32kB

Eg:- 2K pages & 512 frames.

If PAS = 4MB, e = 4B.

calculate N, M, P, f, LA, PA, d, PTS

Sol:-

N = 2K

M = 512.

P = 11 bits ($2K \Rightarrow 2^{1+10} = 11$)

f = 9 bits ($512 \text{ frames} = 2^9$)

LA = 24 bits = (16 MB)

PA = 22 bits.

d = 13 bits (PA = 22, f = 9

d = PA − f = 13)

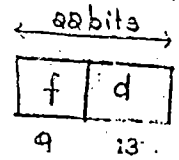
P.T.S (Bytes) = N * e

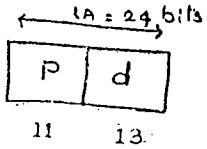
= 2K * 4B = 8 kB

PAS = 4MB

PA = ($2^2 + 20$)

= 22 bits





Performance of paging:-

* main memory access time (mmAT) = 'm' (μs/ns)

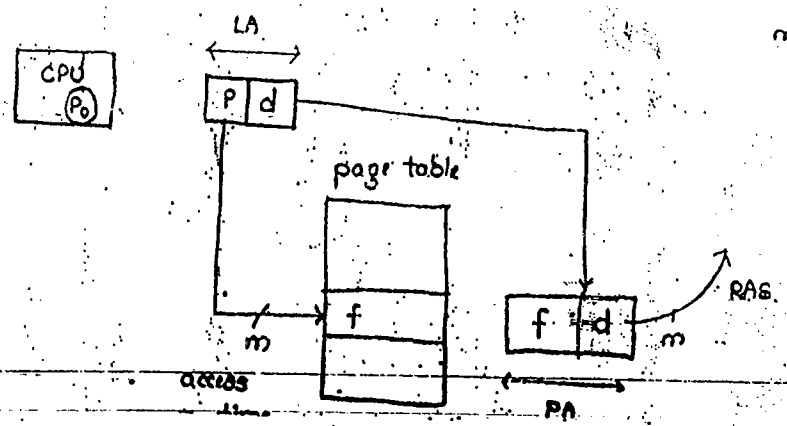
* Effective memory Access time (EMAT)

⇓

Decrease / Reduce of EMAT from 'm' & bring closer to 'm'.

Access Time:

Amount of Time to read (or) write to memory.



(ii) Paging with TLB (Translation look-a-side Buffer):-

If generated LA is not found within the cache (or TLB, it is called "TLB miss". If found, it is called "TLB hit".

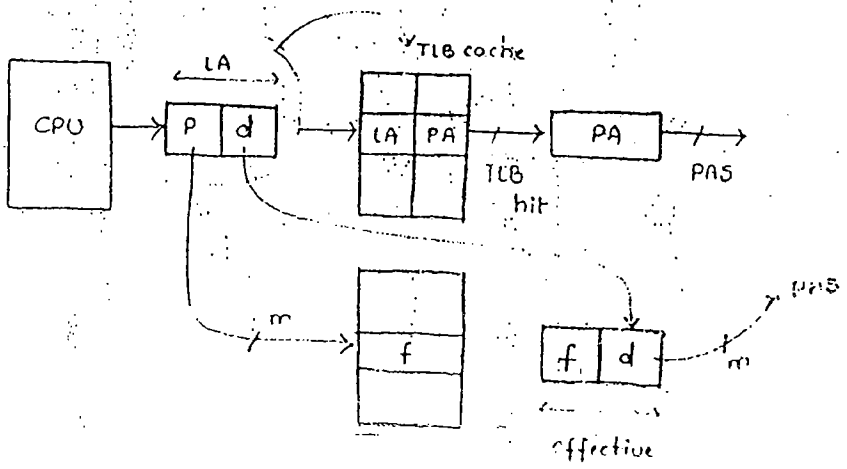
T.L.B Access time = c

where [c << m]

T.L.B page hit ratio (α) = $\frac{\text{No. of hits}}{\text{Total references}}$

T.L.B miss ratio (y) = $1 - \alpha$

TLB is a cache contains list of physical & logical address. If generated LA is present, then it is TLB hit, or it goes to page table & generates physical address



EMAT (SP+TLB) = $\alpha(c+m) + (1-\alpha)(c+2m)$

↓ success ↓ failure.

Eg:- (i) m = 100 ns c = 20 ns α = 90%

EMAT = 0.90 (20+100) + 0.1 (20+200)

= 9(120) + 22 = 130 ns.

(ii) α = 10%

EMAT = 0.10 (20+100) + 0.9 (20+200)

= 12 + 9(22) = 210 ns

(additional time)

Effective memory overhead : = m.

(Simple paging)

SP – Simple paging

$$\text{EMOH}_{(S.P+TLB)} = x(c) + (1-x)(c+m)$$

* By using page table, every time, we require "2m" memory access time 'm' is overhead for minimising. This introduce TLB that access time is 'c' which is less than 'm'. overhead

* In hit rate, Translation require "c+m", if miss occurs it takes c+m+m = c+2m

** Multi-level paging :-

LA/VA = 32 bits P.S = 4 KB.

PT size = 1m B = 4MB

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

e = 4B

1m

Larger page tables are not desired.

Reduction of page table

Increasing page size

⇓

Drawback :-

Increase Internal fragmentation. (wastage of space)

Large :- P.S = 1KB

= 2 pages

Program 1026

= 1024 ≃ 1 page

Small :- P.S = 2B

= 512

512^2 = 0B

Internal fragmentation } must be
P.T overhead } ⇒ minimum.

* LAS / VAS = 's' words
* page Table entry = 'e' bytes.
* page size 'p' = ?

P.T overhead (bytes) = $\left(\frac{s}{p}\right) * e$.

Internal fragmentation = $\frac{p}{2}$

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

To minimize / optimize ⇒ $\frac{d}{dp} = \left(\frac{p}{2} + \frac{s}{p} * e\right)$

(min.) $= 0 = \left(\frac{1}{2} - \frac{1}{p^2} s * e\right)$

$p^2 = 2se$.

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

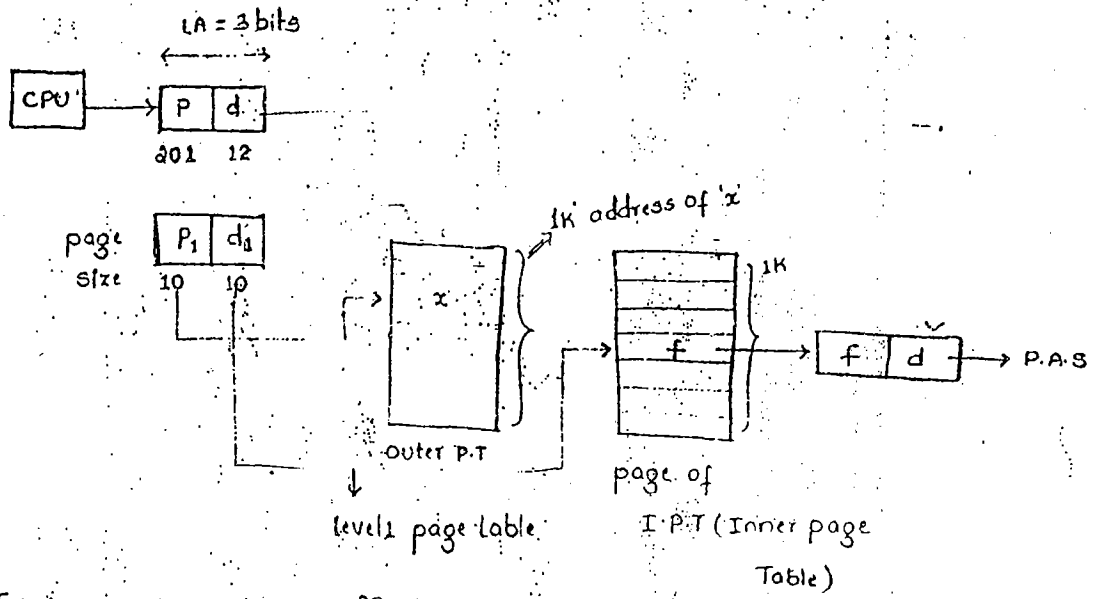
28/07/2010

wednesday

=

Paging :-

* Dividing any address space into equal size units (pages)



Total virtual memory = 2^{32}

Each page size = 4 KB

Total pages are = $\frac{2^{32}}{4KB} = 2^{20} \Rightarrow$ 1M pages

Each page size is 4KB. so, again apply paging on page table of each of a 1K. Total no. of entries = $\frac{1M}{1K} = 1K$

Tooo level addresses = 20 → no. of pages

12 → offset.

Divide again pages of each size = 1K ⇒ It stores address of outer page for storing address it require 10 bits

Performance :-

* Main memory access time = "m"

* Effective access time = m + m + m = 3m

Inner Outer

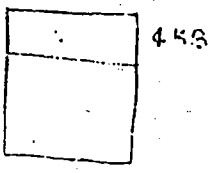
* Effective access time with 'n' level paging = (n+1)m.

For reducing access time using TLB :

TLB hit ratio = 'α'

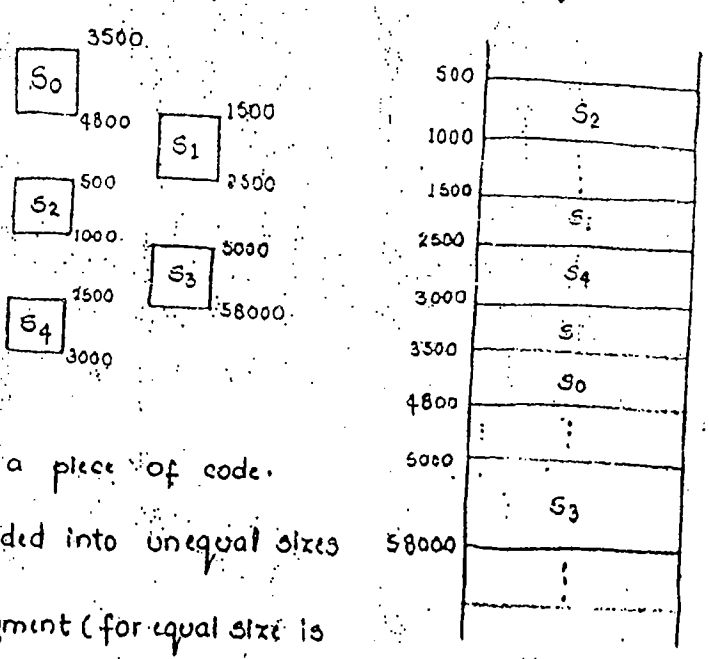
$= \alpha(c+m) + (1-\alpha)(c+3m)$

For n-level pages $= \alpha(c+m) + (1-\alpha)(c+(n+1)m)$



Segmentation

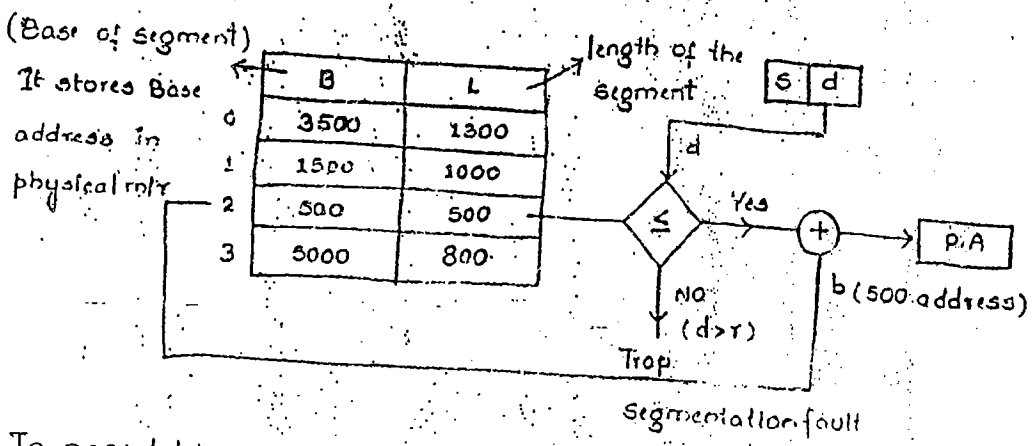
Paging doesn't preserve user's view of memory allocation.



Segment is a piece of code. (program, divided into unequal sizes is called Segment (for equal size is called pages)).

(1) Divide the program into segments (VAS organisation).

(2) Using policies place the segments.



In page table, all are equal sizes. So, no need to compare the required word present within limit (or) not. Here, unequal no. of partitions for checking the offset present in within the limit use magnitude comparator.

Segmented paging :-

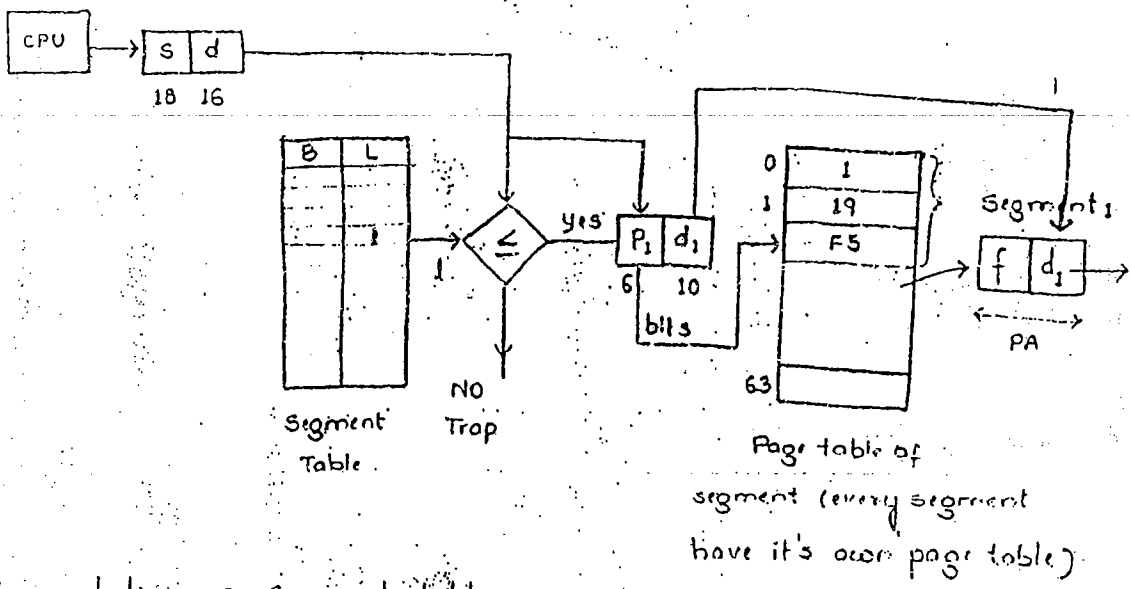
Introducing paging (or) segmentation

Virtual size = 34 bits $\quad <s,d> = <18,16>$ bits

Max. size of seg. size $= 2^{16} = 64$ KB
(↓ offset)

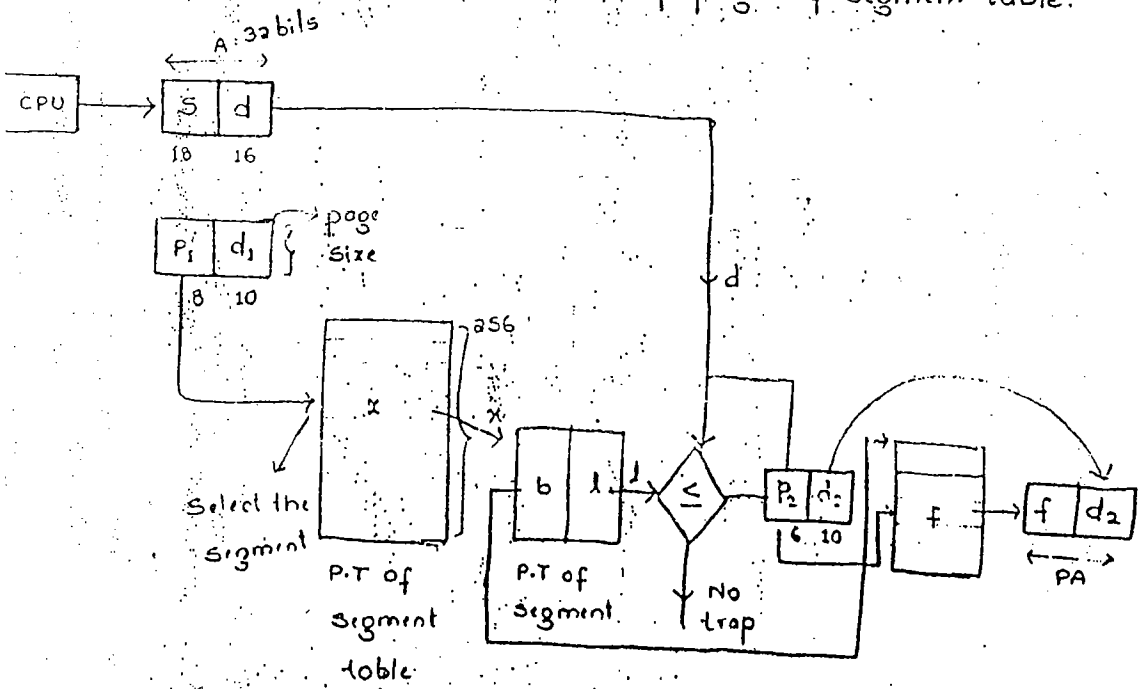
No. of segments $= 2^{18} = 256$ K

Segmentation of Segment Size :-



Segmentation of Segment table:

x = Address of page of segment table.



Paging Vs Segmentation :-

	Internal Fragmentation	External Fragmentation
Paging	✓	×
Segmentation	×	✓

Virtual Memory

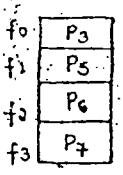
Virtual memory gives an illusion to the programmer that a huge amount of memory available for writing programs greater than the size of available physical memory.

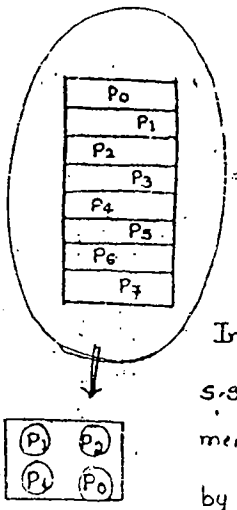
Virtual memory implemented in secondary memory.

Main memory < Virtual memory < Secondary memory

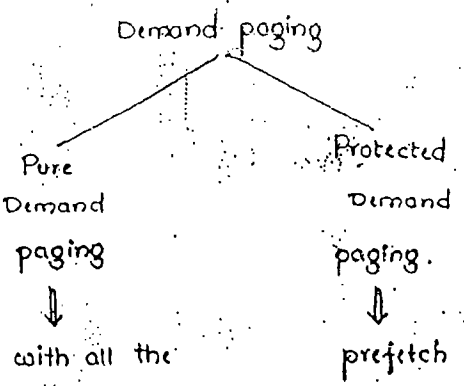
Demand paging :

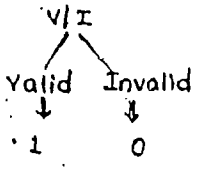
Virtual memory is implemented with demand paging and demand segmentation.

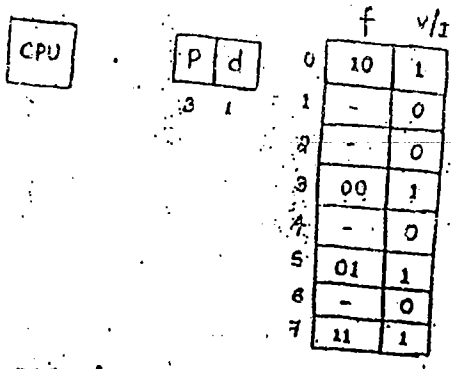




Implemented in S.S. maximum virtual memory is limited by capacity of secondary storage







when page fault occurs, duty of virtual handler :-

* Process is blocked
* Virtual handler → CPU
* V.m handler → Device manager
* Device manager → Device controller
* Device controller → Device buffer.
* D.B → m.m (DMA) transfer the page.
* (i) Empty free is available ⇒ (1) disk operation is enough.
  (ii) No empty free ⇒ page replacement

for this policies

clean (no changed) ⇓

dirty (modified) ⇓ disk operations.

page fault service time ⇒ generally 10^{-3} sec. for memory accessing, it takes $\frac{10^{-6}}{10^{-9}}$ sec.

* Update page table
* Unblock the process

address

Performance

Main memory Access Time (M.M.A.T) = 'm'

P.E.S.T = 's' (s >> m)

Page fault rate = 'p'

Page hit ratio = 1-p

$$\boxed{EAT = (1-p)m + p*s}$$

$$\boxed{E.A.T = (1-p)*m + p(m+s)}$$

Hit ratio is more ⇒ follow the units of main memory access times.

⇒ 1.9999 μs

$$E.T.T = \frac{1}{K}(i+j) + \left(1-\frac{1}{K}\right)*i$$

$$= 1/K + j/K + i - i/K$$

Page replacement :-

) frame Allocation policies :

otal no. of frames = M

lo. of processes = 'n'

emand of each processes for frames = s_i (it depends on total no. of pages of program)

otal demand of process :

$$S = \sum_{i=1}^{n} s_i$$

(Allocation policies)

	s_i	Equal	proportionate	50%
P_1	10	10		5
P_2	25	10		12 only given to
P_3	3 ⟶	10 unjustify		1 so% of resource
P_4	12	10		6
P_5	20	10		10
	70	⇓		

It is applicable for all, which have equal demand.

If $M = 60$, $n = 5$

$\frac{M}{n} = \frac{50}{5} = 10$; $\frac{n}{s} \times m$ (2nd policy)

$P_1 = \frac{10}{70} \times 50 =$ $\quad$ $P_2 = \frac{35}{70} \times 50 =$

(b) Page Reference string :-

Set of successively unique pages referred in the given list of logical a

Logical addresses :-

463, 182, 184, 195, 485, 963, 967, 834, 128, 534, 765, 784, 018, 634, 686

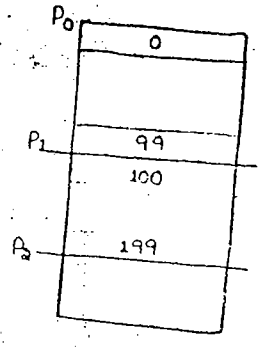
page size = 100ω (consider).

463 => It belongs to page address

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

463 % 100 = 63 → offset.

each page



= {4, 1, 1, 1, 4, 9, 9, 8, 1, 5, 7, 7, 0, 6, 8} // If occured successively, then con

ref string = {4, 1, 4, 9, 8, 1, 5, 7, 0, 6} // '4' occurs again but not successively, so cons only once.

length = 10

No. of unique pages referred in the given list of logical addresses :

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

Eg :-

reference string = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

length = 20, unique = {0, 1, 2, 3, 4, 7}

$n = 6$.

Frame size = 4

page fault = 10

3F → 15

4F → 10

3 frames	4 frames
7 7 7 0 7	7 3 2
0 0 0 2 0	0 4 7
1 0 3 3 1	1 0
	0 1 ...

* After loading page in main memory, how many times a page is referred so far.

: performance of LRU is most closest to optimal.

2	1 => first dropped in
0 =>	2 is replaced
1	1

∴ O.S uses LRU (or) LRU approximations.

LRU Approximations :-

* works like LRU.
* Approximate to the behaviour of LRU

} They are not pure LRU but works like LRU

Reference bit (R) Algorithm :-

* Every page table contains a reference bit within it.

Page Table

P	f	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 far during the present epoch.
R — 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 divided into discrete intervals (Time Quantum) called epochs.

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

	R
P_I	0
P_J	0
P_K	0

= Initially

End of old epoch & starting new epoch has Initial value = 0

	R
P_I	1
P_J	0
P_K	1

page referred in new epoch.

Start searching the page fault (victim) from initial, using ref. bit of LRU approximation is Reference bit algorithm.

If all reference bit values = 1, then this algorithm fails.

So, we introduce a new algorithm called second chance.

(2) Second chance / clock Algorithm :-

Criteria : Arrival time + reference bit

A.T + R

FIFO based algorithm.

Start the search of pages from the A.T. and if it is already referred, then give a second chance to it (i.e., value changed from 1 to 0).

P	f	V/I	AT	R
0	a	1	3	✗ 0
1	b	1	2	✗ 0
2	-	0	-	-
3	d	1	0	✗ 0
4	k	1	1	✗ 0

(3) Enhanced Second chance: Avoids unnecessary page back.

(not-recently used)

Criteria : (R) + (m) (modified bit)

+

(dirty bit)

modified bit => checks whether the contents of page are modified (or) not.

0 →

1 → write back.

R	m	
0	0	not R, not m
x 0	1	not reffred but modified
1	0	Referred but not modified
1	1	Referred & modified

R = 0, M = 1 :-

	R	m	R	m
Suppose P_i →	0	1	0	1

It is referred in old epoch and it is not referred in new epoch but modified.

when page fault occurs consider the value of "00" 0 → LRU

we look for '01' combination next since, it represents LRU

Then '10' is considered & at last '11' is considered.

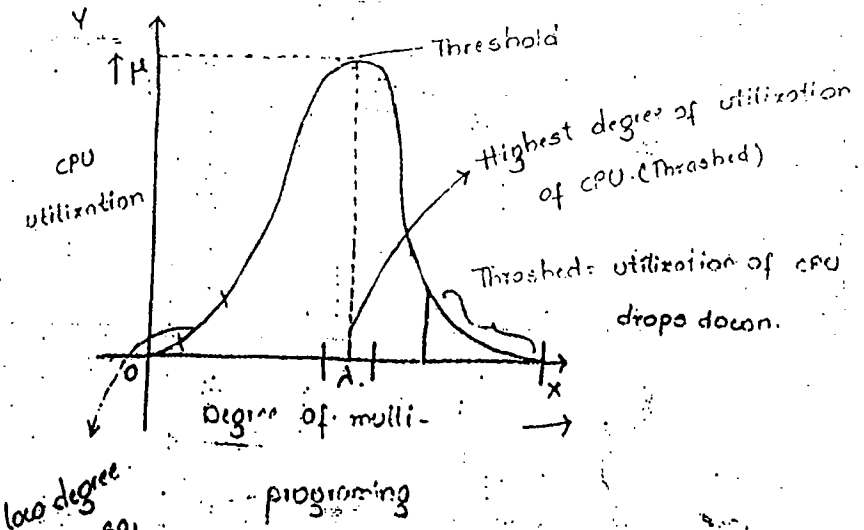
Thrashing :- (High paging activity) (page fault rate).

* Excessive / High page fault rate is called Thrashing.

* It is also an undesirable state of the system like deadlock.

Reasons for Thrashing :-

Primary	Secondary
(1) Lack of memory (frames) i.e., if frame allocation to a page get reduced.	(1) page replacement algorithm. Drawback: fragmentation
(2) High degree of multiprogramming	(2) page size: small (2B) (512 pages) ; large (1024B) => 2 pages => 0 page faults less pages (then less page faults) when reduces thrashing
	(3) program structure.



Thrashing control strategies :

I. prevention & Avoidance

* Controlling degree of m.p in and around "μ".

II. Detection & Recovery.

* low cpu utilization, along with high degree of m.p.
* paging disk utilization must be high.

Recovery :

* By applying concept of "process suspension" (swapping out some processes to secondary memory)

Page size :

Temporal (Time) → Thrashing

Spatial (space) → Internal fragmentation

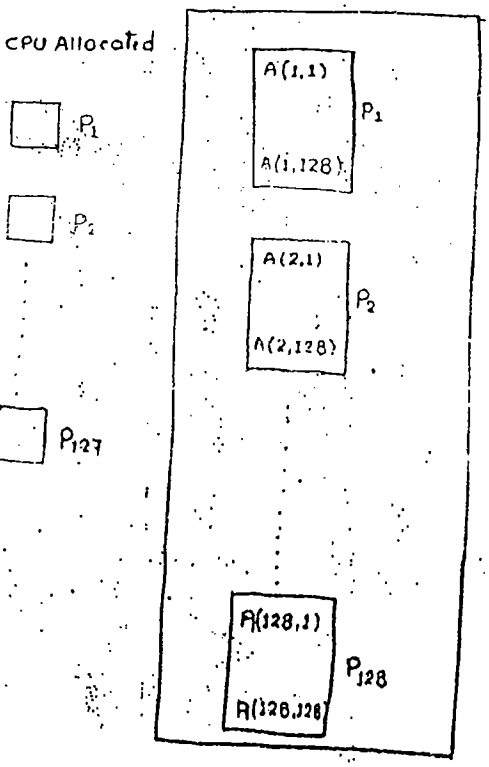
Program Structures :-

int A(1...128, 1.... 128)

page size = 1280

i) for i ← 1 to 128
   for j ← 1 to 128
   A[j,i] = 5

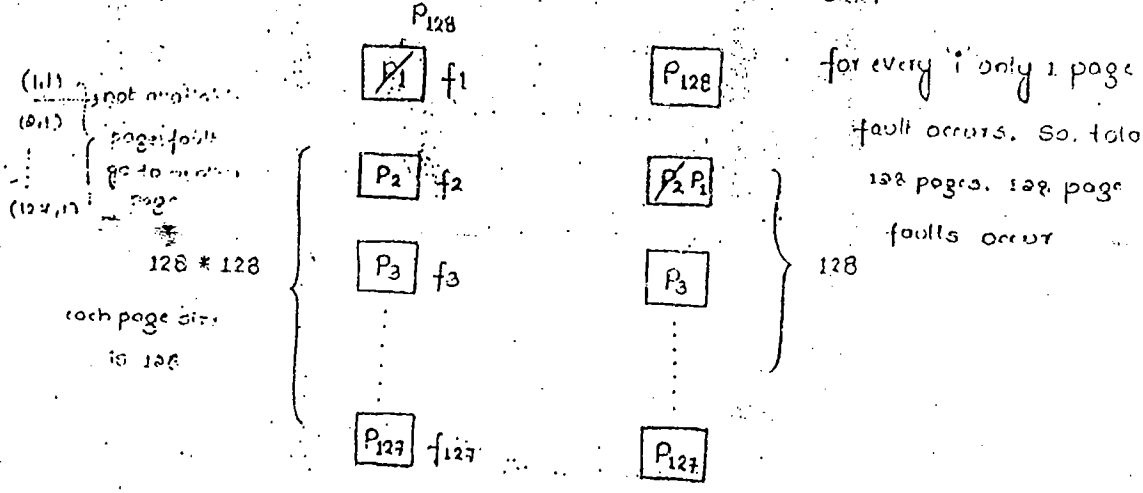
ii) for i ← 1 to 128
   for j ← 1 to 128
   A[i,j] = 5



(Q) If there are a set of processes (from 1 to 128) to be executed. But, If the CPU allocates only 127 processes to get executed. Then, how the CPU allocation processed for the above two conditions.

Sol:- Column major order ← A[j,i] A[1,1]

A(i,j) → row major order



working set model :-

* Based on locality of reference.

```
main()                          g()
{                               {
   f()  — 5K.                      h()  — 8K.
}                               }        locality (dynamic)
f()                             h()
{                               {
   g();  — 10K.                    — 15 K
}                               }
```

Total = 38 K

By monitoring the locality, main block requires 15 K, but memory gives 15 K based on locality, it takes frame no's

By knowing, no of frames required based on Guess/ Estimate reference string and divide.

Objective :-

* The main objective is to reduce page fault rate.

* Locality may be a function, block, class (or) module.

* Hold the pages of the locality at which the present function is performing.

Guess / Estimate : Estimating the ref. string Δ in each function.
size

Reference string (pri):

7, 6, 8, 9, 6, 8, 9, 7, 12, 16, 20, 22, 18, 20, 12, 23	28, 29, 34, 38, 27, 28, 38, 29, 38, 34 ↓t
main() f()	g()

Working set window (WSW):

* Set of unique pages referred in the given list of reference string during the past 'Δ' references.

⇓

Integer (Guess)

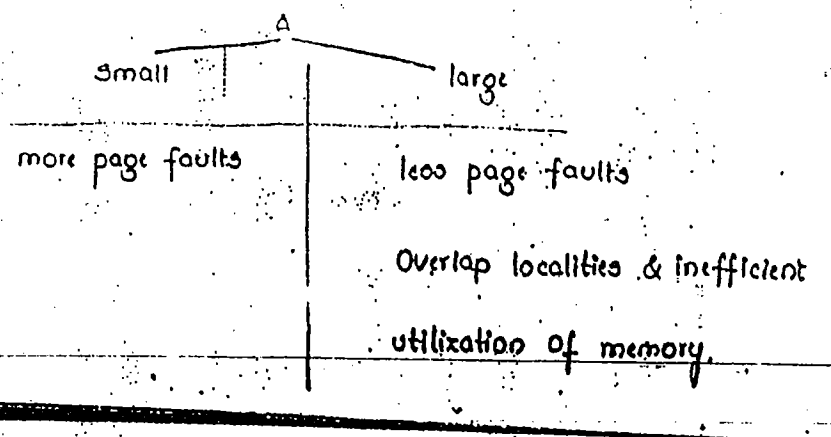
Let Δ = 10 (Guess) ⇒ no. of integers in g() = 10 (length)

WSW9 = {23, 27, 28, 29, 34, 38} = 6.

⇓

Unique reference string from the Set of g() function.

Success will depend on the value of Δ.



* If one guess the value of Δ to be small, then there is a chance occuring more page faults.

Eg:- If Δ = 10

WSWS = { 23, 28, 29, 34, 38, 34, 28, 23, 29, 38, 34 }

page faults = 6.

If Δ = 5

WSWS = { 28, 23, 29, 38, 34 }

page faults = 5.

If Δ = 5

WSWS = { 23, 28, 29, 34, 38 }

page faults = 5.

∴ for the same set of reference string, if Δ = 10 (more no. of integers are consider, then page faults = 6. other than if Δ = 5 ; it have page faults = 10.

* Let $P_i = 1 \ldots\ldots n$

$|\cos\omega s|_i$ = where $i = 1 \ldots\ldots n$

Total demand $(\theta) = \sum_{i=1}^{n} (\cos\omega s_i)$

Total frames available = "m"

* If m = θ (no thrashing, system is perfectly balanced).

* If m >> θ (no thrashing).

(scope of increasing degree of multiprogramming).

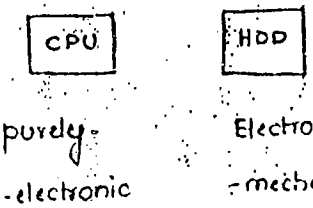
* If m << θ (Thrashing)

1000 10,000

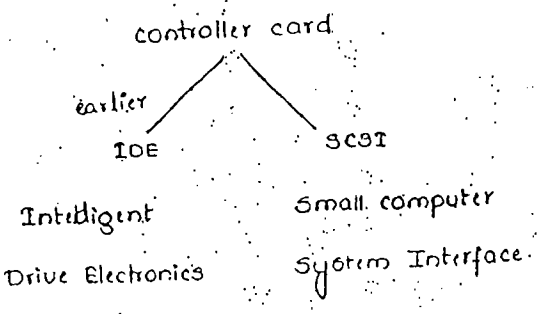
File System & Device management

Interface :

Every secondary / Tertiary media must have their ocon file system.



(1) We need a controller interface as controller card / chip. ⇒ Hardware requirement



(2) Software requirement :

* Device driver

3) Device Independent software :

* It is called as file system. (Third party device drivers).

Files & Directories :-

file :- Collection of logically related set of records of an entity.

* It is considered for data structure.

* **Data structure :-**

  (1) Definition

  (2) Representation / Implementation :

  * Flat structure
  * Record structure
  * Tree structure

  } ⟹ file structures.

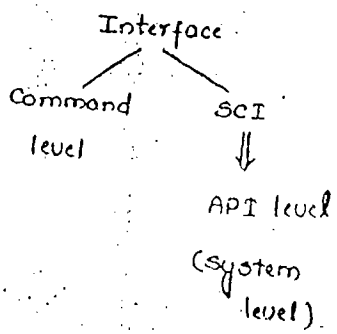
  (3) Operations :

  * Create a file
  * Open a file
  * Read / write / append / seek / modify / Truncate
  * close a file
  * Delete a file

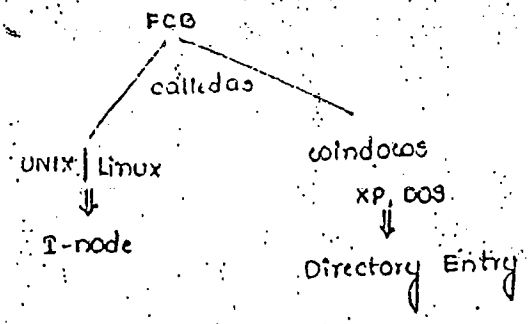
  (4) Attributes :-

  * name, extension of file
  * Type of file
  * Size of file
  * owner
  * permission
  * mode of access (sequence / random)
  * Time & Date stamps
  * link count.

  } Stored in file control block (FCB)



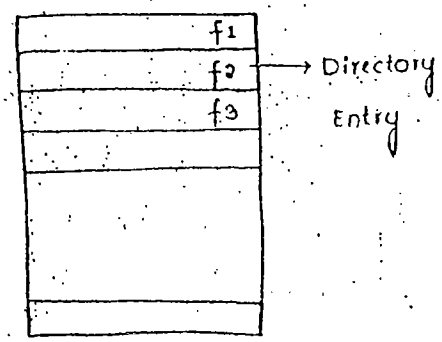
* Attributes of a file are stored in file control Block (FCB).



Directory :-

* It contains information about files i.e., meta data of files.

Abstract view of directory (meta data of files)



1 Directory Entry per file.

Organisation of Directory Structure :

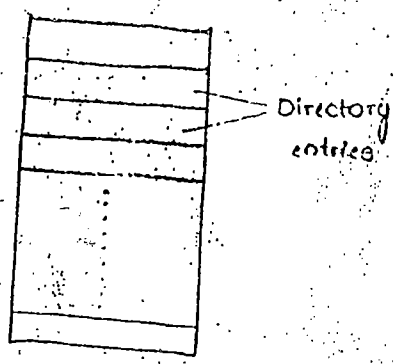
* Single level directory
* Two level directory
* multi level / Tree Structures
* Acyclic Graph.

31/07/2010

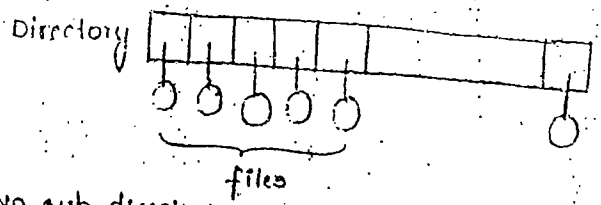
Saturday

Directory Structures :-

Abstract view of Directory



(a) Single-level Directory structure :-



* No sub-directories are present (one entry for each file)
* All the files are managed by one directory.

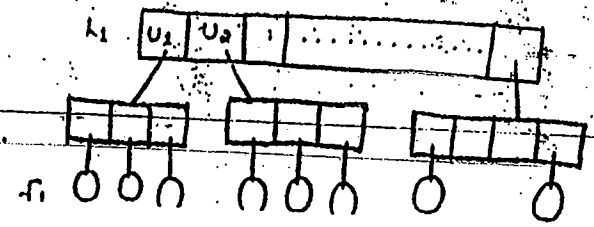
Advantages :

* Simplicity
* Searching
* Name conflicts

Disadvantages :-

* Only one directory, no sub-directories
* Two different users, simultaneously can have same filename

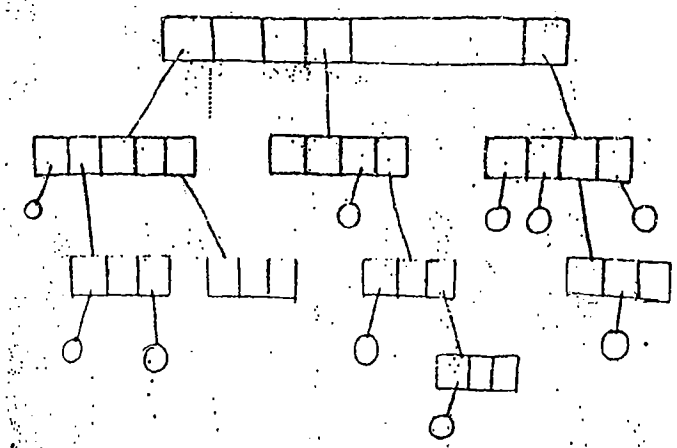
(b) Two-level Directory structure :-



Advantages :-

* Creating different files with same name (flexibility)

3) Tree-structured directory :-



Drawback :-

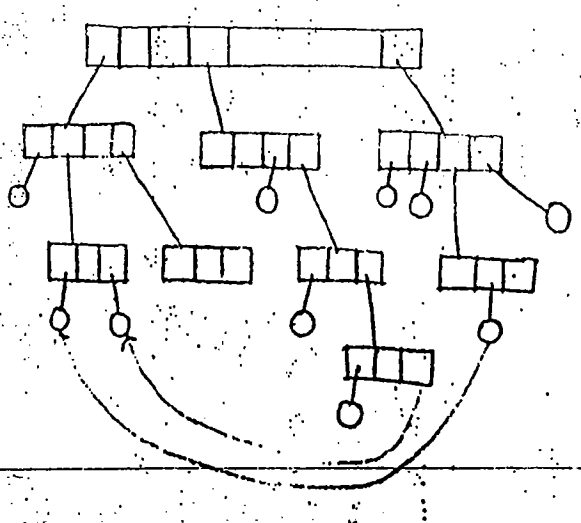
* File sharing :-

* Sharing by duplication has drawbacks of :-

    * Inconsistency

    * wastage of space

To overcome duplication in file sharing, we have the concept of file sharing with links called "Directed Acyclic Graph"

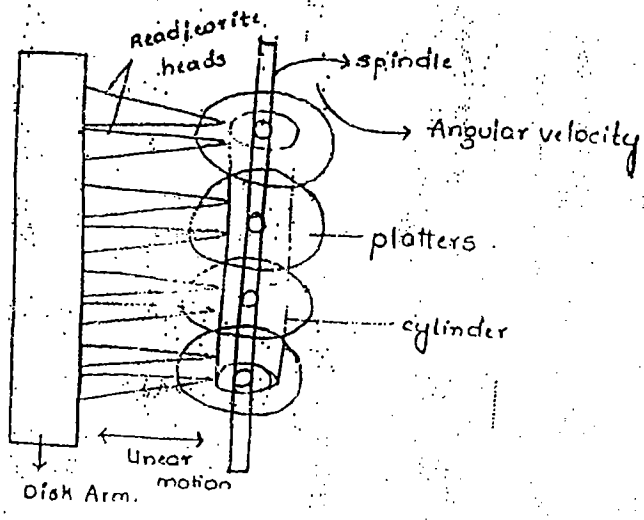


Device characteristics:

* Major component in secondary storage is disk.

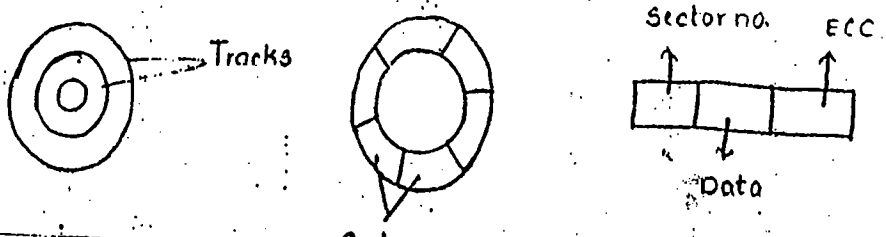
Physical and logical structure of Disk :-

Physical structure / geometry of Hard disk :



* Hard disk consists of set of platters, with a spindle.
* Each surface of platters is associated with Read/write heads.
* Two Types of motions are supported :
    * linear velocity (move forward & backward) with disk armature
    * Angular velocity with disk spindle.
* Tracks are divided into sectors.

Sectors :-



Sectors can be represented by three fields :-

* Sector no.
* Data
* Error correction code (ECC).

Error Correction code (ECC):

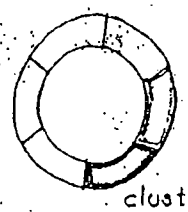
* To detect the deadlock. By reformatting the bad sector, some sectors can be made good sector.

Cylinder :-

A group of same track number.

Cluster :

A group of one (or) more adjacent sectors is a cluster. A cluster is one of the unit of Ilo transfer.

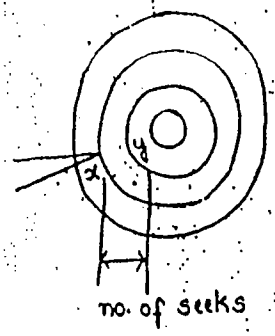


cluster

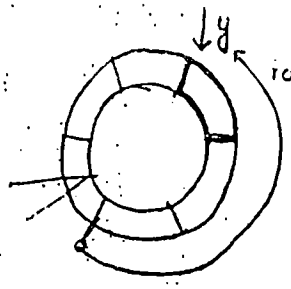
Ilo Transfer :-

The amount of time taken to a sector to make a move from present area to desired area is called seek time.

Seek time:
time taken to travel from 'x' to 'y' (or) linear motion from x to y



no. of seeks



rotational latency

time = $R/_2$ (average)

Seek time + (Rotational) latency time

$\Downarrow$

$\frac{R}{2}$ (on an average)

where R = Time for one rotation.

Transfer time (Transmission Time) :-

* The transfer time depends on two factors :
  * Track size ((S) bytes)
  * Rotation rate (RPM)

Transfer time :
* Read the data to sector

Sector size = 'x' bytes ($x < s$) i.e., sector size < Track size.

Rotation rate = 'r'.

Time for one rotation = 'R'

$$R = \frac{60}{r} s$$

3600 r = 60 s

1r = ?

$\Rightarrow \frac{60}{3600} = \frac{1}{60} \Rightarrow R = \frac{60}{r}$ seconds

If $\frac{60}{r}$ s $\Rightarrow$'s' bytes

? $\Rightarrow$ 'x' bytes (read)

$$\Rightarrow \frac{x * 60}{r \times s} \times \text{sec} = \frac{x * 60}{r}$$

bytes

Disk I/O Time :-

= (Seek Time) + (Latency Time) + (Transfer time)

$\Downarrow$

$$= \text{S.T} + \frac{R}{2} + \frac{x * 60}{r \times s} \text{ (sec)}$$

$\frac{60}{r}$ — s

? — x

$\Rightarrow \frac{x * 60}{r * s}$

Effective Data Transfer Rate (DTR) (Bytes/sec) :- no. of bytes in 1 sec.

Assume Track size = 's' bytes

RPM = 'r'

Rotation time = $\frac{60}{r}$ sec

$\frac{60}{r}$ sec ——— 's' Bytes

1s ——— ?

$$DTR = \frac{r * s}{60} \text{ bytes/sec}$$

Eg:- consider a hypothetical disk with :-

16 platters
↳ 2 surfaces
↳ 2 Tracks
↳ 512 sectors
↳ 2KB

Disk seek time = 30 ms

RPM = 4000

Then calculate (i) capacity (ii) I/o time (iii) DTR

Sol:-

(i) Total no. of sectors = $2K \times 512$

capacity of one surface = $2K \times 512 \times 2KB$

$= 2^{11} \times 2^9 \times 2^{11}$

Capacity of two surfaces = $2^5 \times 2^{11} \times 2^9 \times 2^{11} \times 2^{36}$

= 64 GB

(ii) I/o time = seektime + latency time + Transfer time

$$\frac{R}{2} + \frac{\alpha * 60}{r * s} \text{ (s)}$$

$$R = \frac{60}{4000}\,s \Rightarrow \frac{R}{2}\ (\text{latency time}) = \frac{60}{8000}\,s.$$

Transfer time :-

$\frac{60}{4000}$ s — 1 MB

? — 2 KB

$$\Rightarrow \frac{60 \times 2\,KB}{4000 \times 1\,MB}\,s. = x\,(s)$$

$$\therefore \text{I/o time} = \underbrace{S.T + L.T + T.T}_{\downarrow} \quad \frac{R}{2} + \frac{x \times 60}{7200}\,(s)$$

$$= 30\,ms + \frac{60x}{8000}\,s + x$$

$\frac{80}{8000}\,s$ x.

(iii) DTR :-

$\frac{60}{4000}$ s — 1 MB

1 s — ?

$\frac{4000 \times 1}{60}$ mB/s

Logical structure of Dion (formatting process) :-

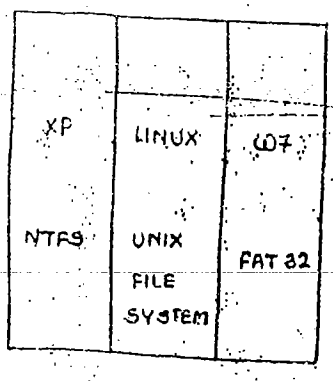
Popular file systems :-

* NTFS
* Solaris
* FAT 32

} popular.

* UFS
* ZFS

fdisk ⇒ create, delete, & edit the partitions



partitions (volumes)

(C:; D:; E:;)

Partitions

- Primary
  - * Bootable (stores O.S onto it) & userdata
- Secondary (Extended)
  - * Non-bootable ⇩ only userdata get stored.

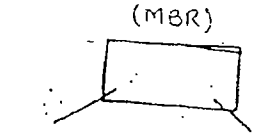
In the partitioning of a system, there must be atleast 1 primary partition, and the rest may (or) may not be of secondary.

Computer system, which supports multiple O.S to boot is called multi-boot computer.

File format system:

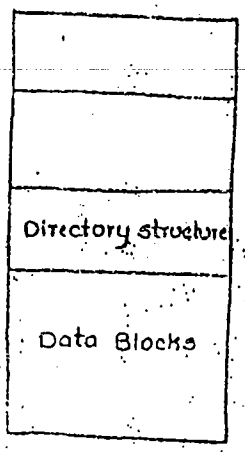
We can select any file format with different O.S's.

Master Boot Record (MBR)



- Boot process (Booting the selected O.S)
- partition table (It represents the complete picture of partition) (only one entry for the partition).

Partition Structures :-



BCB block is empty => If there is non-bootable partition.

BCB can be represented in different terms :-

* Boot Block (In UNIX)
* Partition Boot Sector (In NTFS) (It contains boot of that sector).

PCB gives information about other blocks.

PCB can be represented in different terms :-

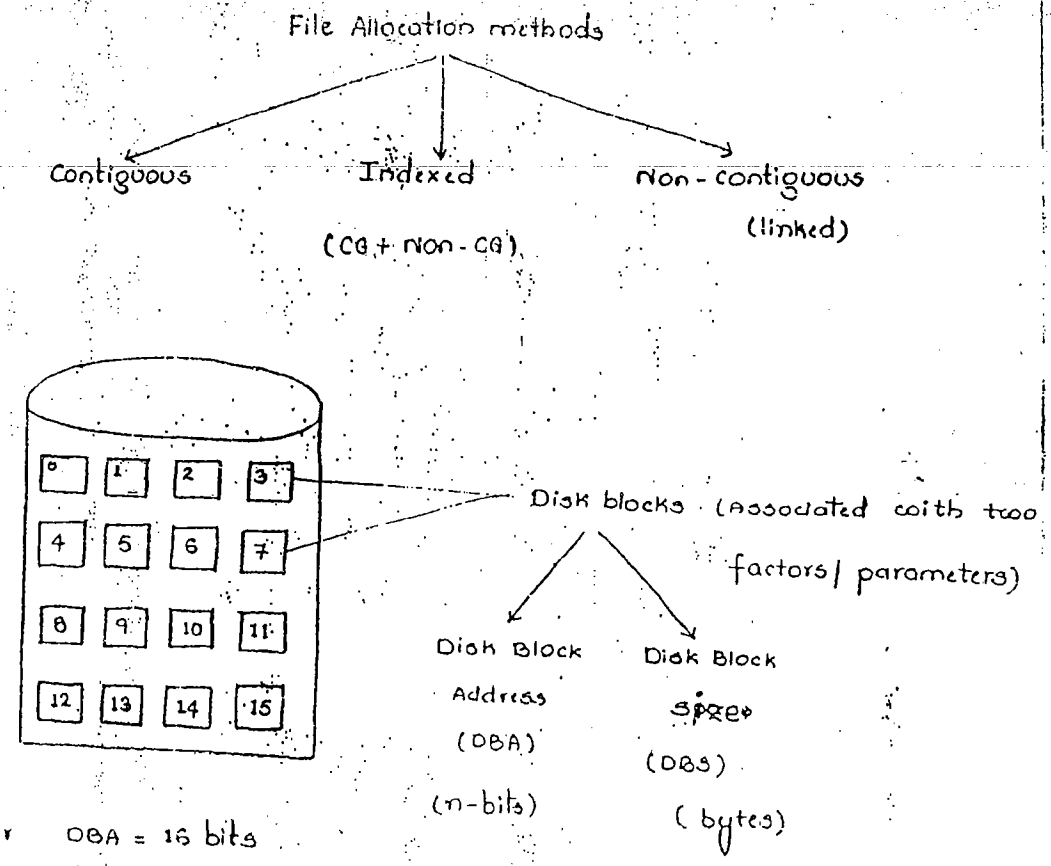
* Superblock (In UNIX) (gives information about others)
* Master file Table (M.F.T) (In NTFS)

Disk space is being organised in blocks.

Block => unit of Allocation.

Allocating the disk space methods => file Allocation methods.

Disk File Allocation methods (or) Disk Space Allocation strategies :-



onsider DBA = 16 bits

DBS = 1 KB

Then, maximum file size (in bytes) = 64 MB

⇓

depends on/limited by

disk size

$$\text{Disk Size} = 2^{16} \times 1\ \text{KB}$$

$$= 64\text{K} \times 1\ \text{KB}$$

$$= 64\ \text{MB}.$$

(1) Contiguous Allocation :-

first block of disk used by file

directory entry

filename	starting DBA	file size (no. of blocks)
test.c	5	4
temp	10	6

(5, 6, 7, 8 blocks)

(10, 11, 12, 13, 14, 15 blocks)

Atleast 'm' contiguous blocks require.

Performance :-

(1) Fragmentation :-

Internal

External

(not contiguous, but bifercated with memory)

If last block files are not fully utilised, then it goes waste, which represents internal fragmentation.

(2) Increasing file size :

May / may not be possible

Inflexible

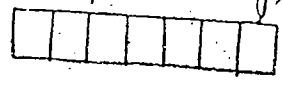
(3) Type of Access :

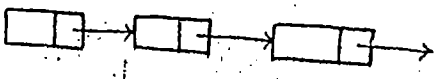
Sequential

Random

Both are supported

Sequential (Array)





Random (linked list)

Since, it supports random access, it is faster.

(2)

Non-Contiguous/ Linked Allocation :-

Filename	Starting DBA	Ending DBA	
test.c	4	3	(4,9,6,3)
temp	5	2	(5,1,2)

Performance :-

(1) Fragmentation :

Internal ✓ External ✗ (Any block can be utilised)

(2) Increasing file size :-

File size increment is possible, as long as free blocks are available.

(3) Types of Access :

Sequential access only possible. No random access. So, it is slow.

Drawbacks :

* Some disk space is consumed for storing pointers, that addresses next instruction block.

* vulnerability of links :

↓ expose danger ↓ breakage of links

↓

Then, file gets truncated.

Reason for storing ending DBA :-

* We store ending DBA, (along with starting DBA), because for the following :-

  * Creating a new pointer done fastly.

    i.e., To extend a new file along with the existing, it can be made faster.

  * Checking file system consistency. (scandisk)

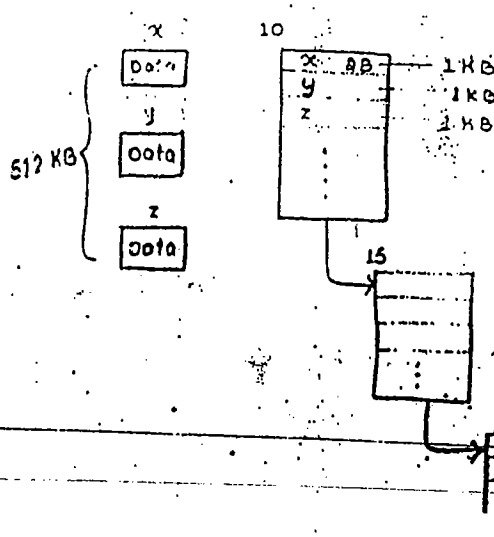
    i.e., checking whether file is fully accessed (or) not.

(3) Indexed Allocation (Contiguous & Non-Contiguous) :-

Filename	I-Block
test.c	10
temp	15

* A file can be allocated in both contiguous and non-contiguous order.

* Eg. :- DBA = 16 bits , DBB = 1 KB



DBA = 16 bits = 2 Bytes

$\frac{1KB}{1B} = 512 KB$ $\quad 2^{16} = 64 K$

64 K x 1 KB = 64 MB

512.

Using 1KB Index, we can represent 512 KB no. of blocks.

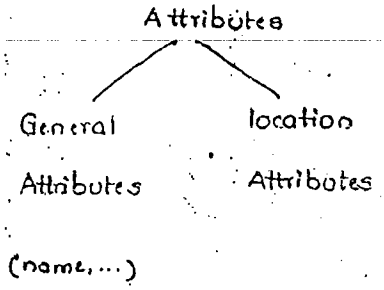
04/06/2010

Wednesday

UNIX - I-node structure

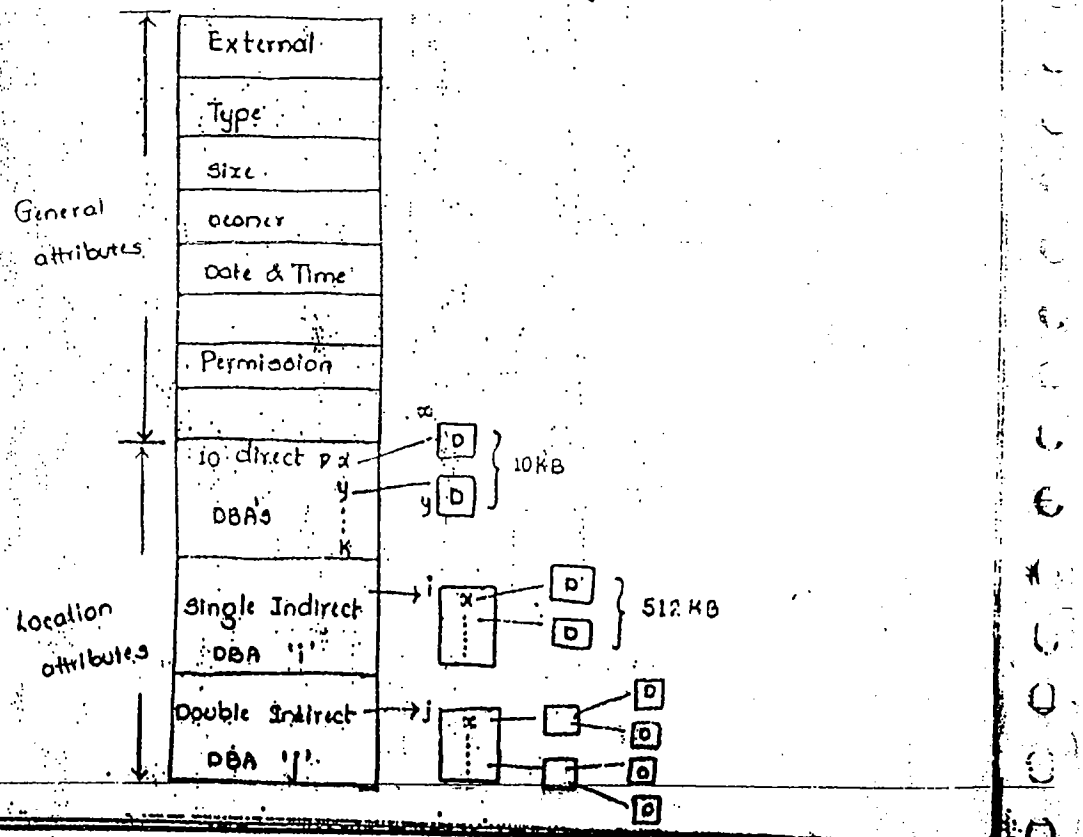
Directory entry consists of two fields :

* filename
* I-node (contains Attributes)



filename	I-node (23)
test.c	23
temp	40

I-node(23) => block of memory



* Maximum file size is limited by the maximum disk size = 64KB

Single Indirect DBA :-

DBA = 16 bits DBS = 1KB

2^{16} = 64K

64K * 1KB = 64MB

512 * 10 = 522 KB = 0.522 MB

Double Indirect DBA :-

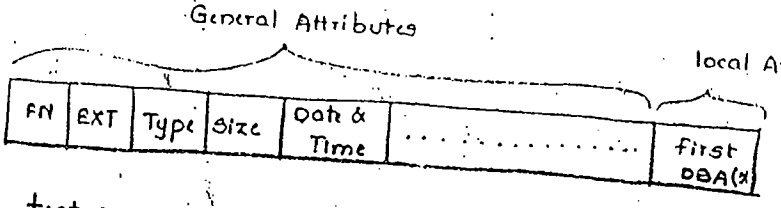
512 * 512 KB = $2^9 \times 2^9 \times 2^{10}$

= 2^{28} = 256 MB

Total size = 256.522 MB (logical) => not considered because max. file size is only 64

DOS directory Implementation :-

* DOS directory implementation is complex than that of UNIX.
* All attributes are embedded in the directory entry itself. There is no I-node in DOS implementation.



test.c ..... Directory entry x → first block address of data.

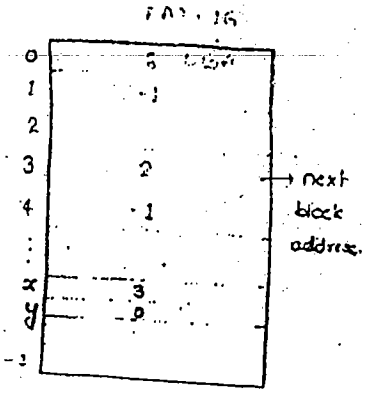
* For the remaining data addresses to represent, we use file Allocation Table (FAT).

No. of Entries in FAT = No. of blocks on disk. 'n' blocks = 0 to (n-1)

* Only the first block address of data is being indicated as a field, and the remaining data addresses are stored in FAT table, and referred whenever required.

* To know the (block) address of data, move to 'x' address on FAT. It represents the next data address. If there is last address, then it is represented by "-1" (i.e., it is last data)

* FAT entry represents ⇒ address of disk block.



Tabular linked allocation

DISK FREE SPACE MANAGEMENT

Eg :- consider disk = 20 MB

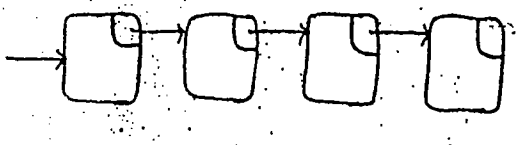
DBA = 16 bits DBS = 1 KB

∴ Total no. of blocks = $\frac{20\text{MB}}{1\text{KB}}$ = 20K blocks

Initial, 20K blocks are free.

Approaches, to keep track of these 20K blocks :-

(1) Free linked list :-



If any file requires the block, it deletes the particular size (eg. 10) and allocates it to the file.

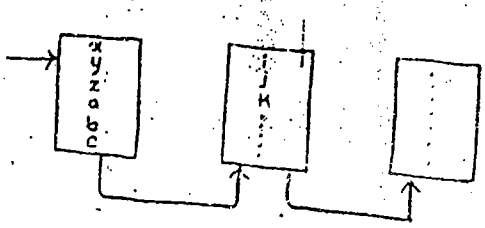
(2) Free list :

* The addresses of the files are stored in linked lists.

1 block ⇒ 512

? ⇒ 20 K

$$\frac{20K^2}{512} = 40$$



* If the disk is almost free, it requires 40 blocks.
* If the disk is almost full, it requires 1 block.

(3) Bit-map method :

Associate a binary bit with each block
- 0 — block is free
- 1 — block is in use.

1 1 1 0 0 0 1 0 1 1 0 0 1 0 1 1
1 0 1 0 0 0 1 1 1 1 0 0 1 0 1 1 1
1 1 1 1 1 0 0 1 0 1 0 1 1 1 0 1

Size of bit map for the blocks = 20 K bits.

1 block can store = 8K bits

? — 20 K

$$\frac{20K}{8} = 2.5$$

≃ 3 blocks.

It requires searching time (more) for a file,

So it is slower than free list.

Block size ≤ $2^{0.9}$.

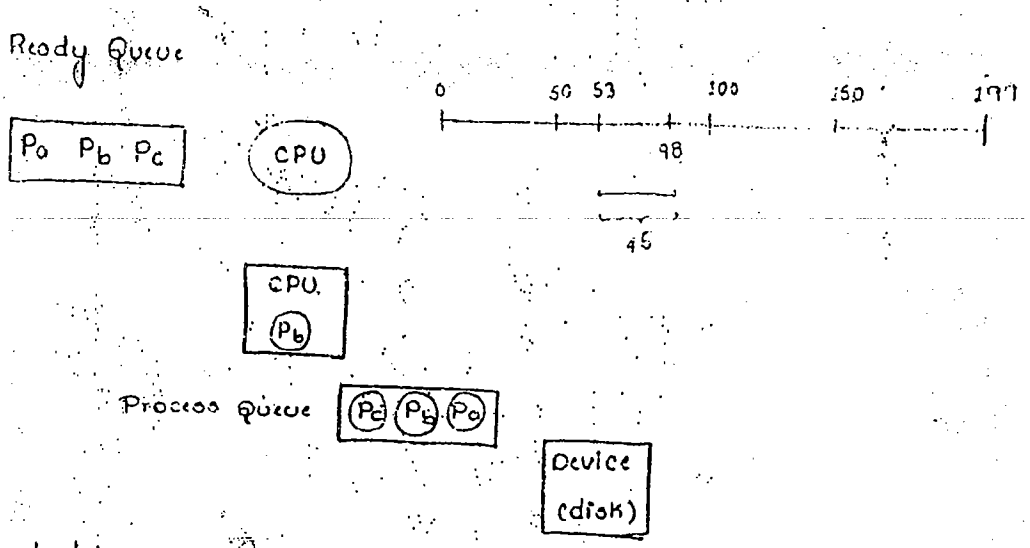
S - DBS

B - Blocks

D - DBA

P - Freeblocks.

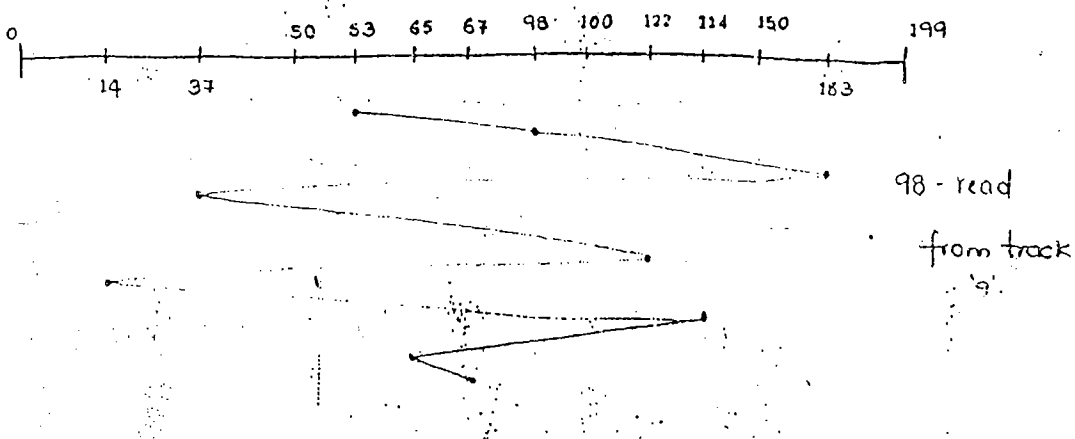
Disk scheduling :



Disk scheduling is used to represent which process is serviced next, when a process completes it's operations.

(1) FCFS :

Requests : 98, 183, 37, 122, 14, 124, 65, 67

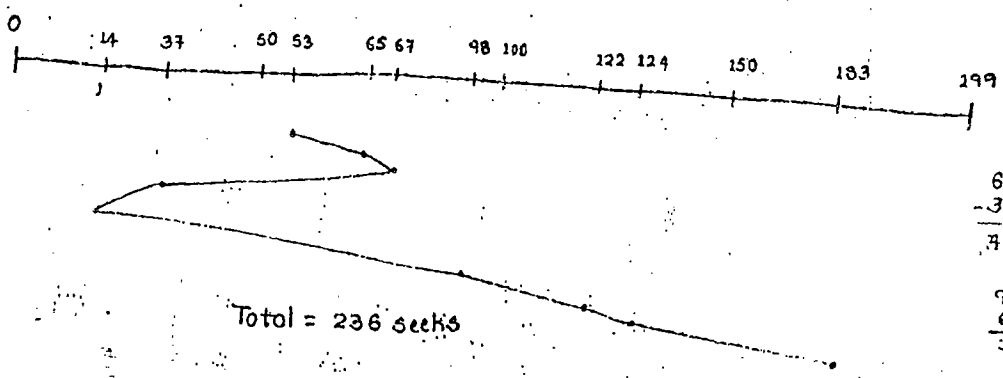


Total = 640 seeks

To reduce no. of seeks, we have SSTF.

(2) Shortest Seek Time first (SSTF) / Nearest Track Next (NTN) :-

(Greedy method)

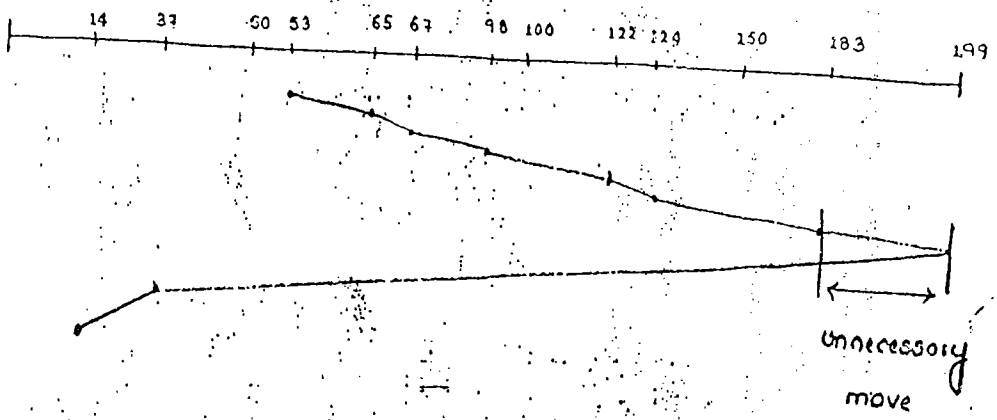


(3) SCAN / Elevator algorithm) :-

* Read-write head ⇒ represents directional scan.
* scanning through the request encountered in certain direction.

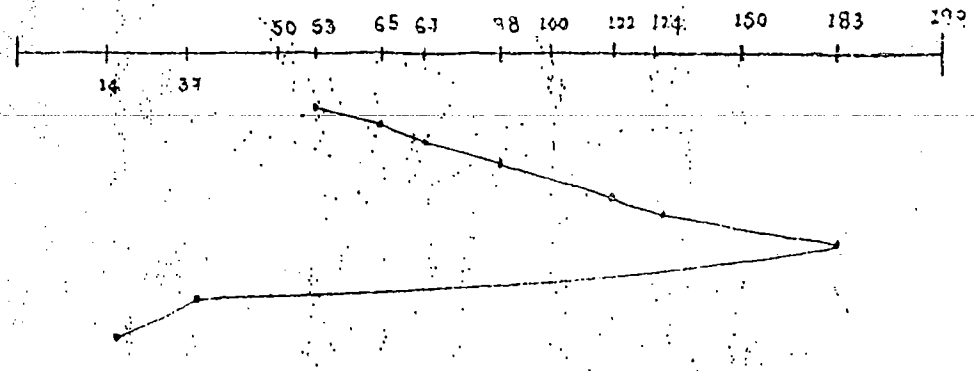
Drawback :

* It moves to the last track unnecessarily.
* Unnecessary causing starvation, by pending the other track in other direction.



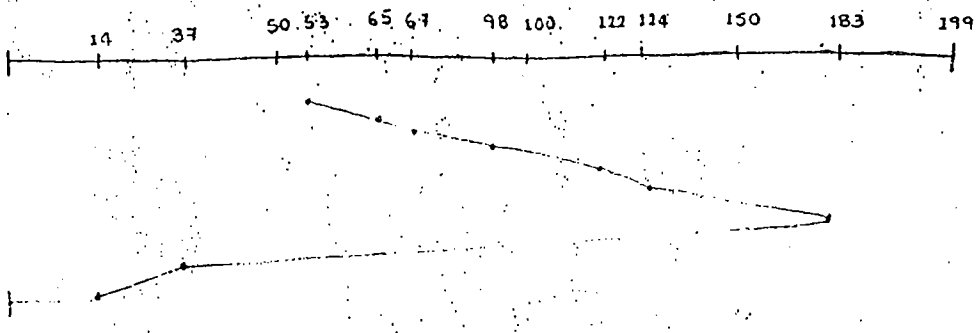
(4) Look :

Look up to last request pending and take reverse turn.



(5) Circular Scan (c scan):

Eg:- Disk



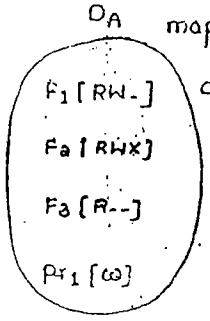
Security Vs. Protection

Security :

* Making system secure from external threats (virus, worms, trojans etc)
* Security / protection has difference of authorised & unauthorised users.
* Protection mechanism is implemented using "domain() method"
* Protection Domain (objects, Rights)

  ⇓

  refers either user/process (with Rig. permissions)

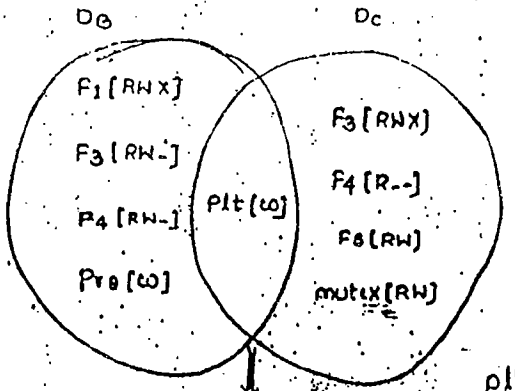


Protection :-

Internal threats
(system level)

Security :

External threats



plt ⇒ platter...

Overlapping domain

(1) Protection Domain matrix :

(Printer)

	F_1	F_2	F_3	F_4	F_5	Pr_1	Pr_2	Pr_3	Plt	mutex
D_A	RW-	RWX	R--	-	-	W	-	-	-	-
D_B	RW		RW	RW			W			
D_C			RW	R	RW					RW

n x m

n → no. of domain

m → no. of objects

entry → permission.

- They support direct access. (so, they are faster)

Drawback : space wastage (called sparse matrix)

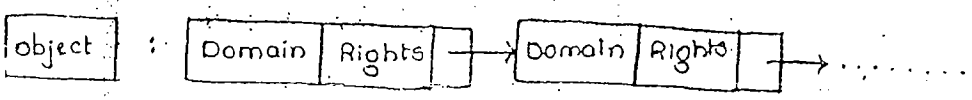
⇓

most are null entries.

To overcome

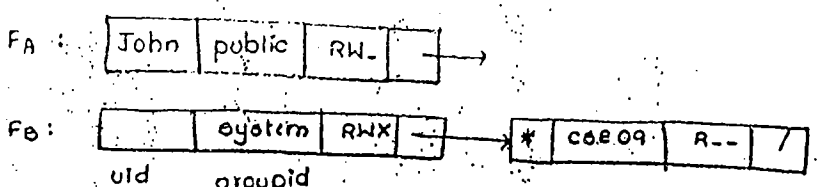
⇓

a) Access Control list (A.c.c) mechanism :

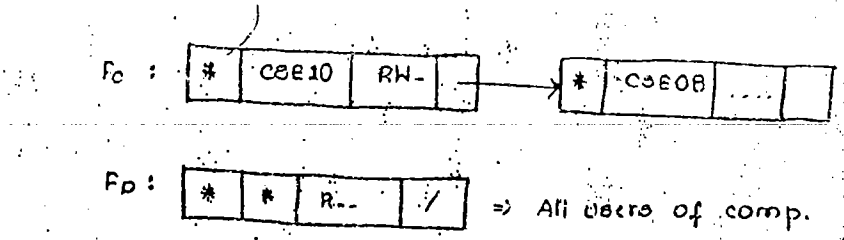


linked lists headed by objects

used in : UNIX / LINUX



file used by all cse people.



⇒ All users of comp. & use people

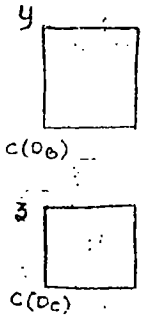
(3) Capability list (c-list):

Type	Rights	Name/Address
FILE	RW-	F_1
FILE	RWX	F_2
FILE	R--	F_3
PRINTER	W	Pr_1

→ stored in fixed place

Capability (D_A)

windows

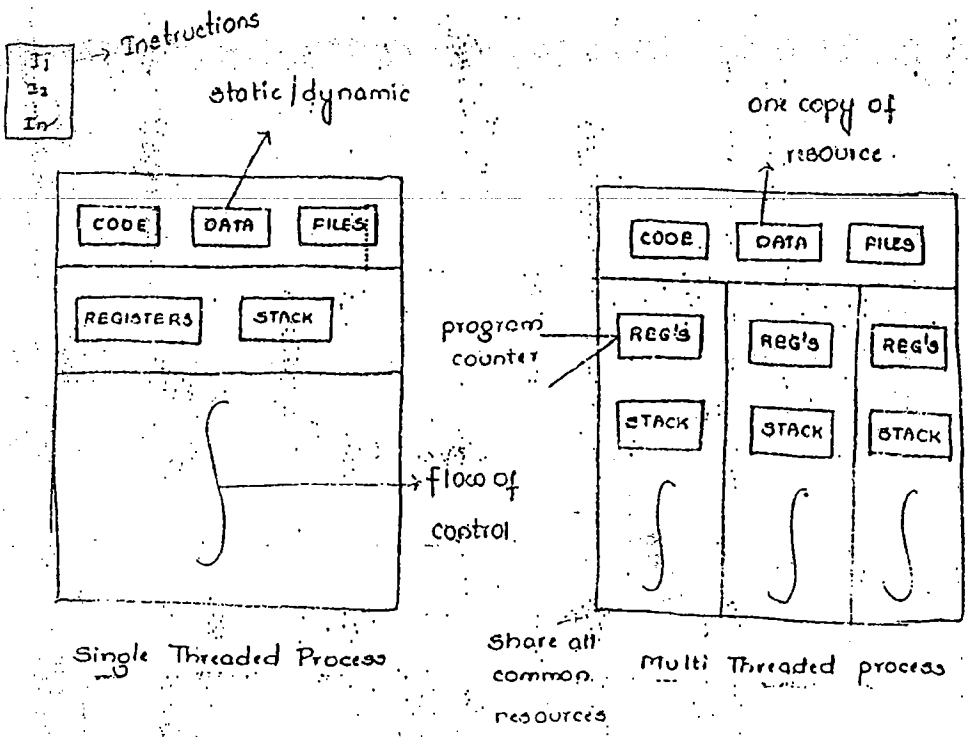


protection mgmt follows

c-list

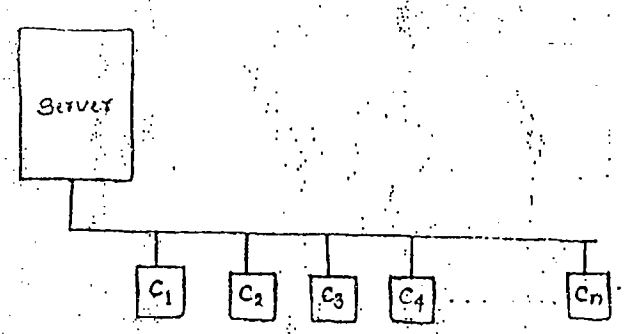
Domain	Address (capability)
D_A	x
D_B	y
D_C	z

Threads & Multithreading



Thread - light weight process.

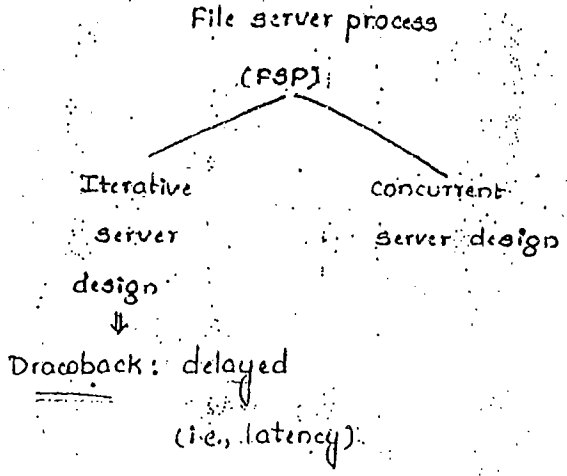
LAN (File Server process) :-



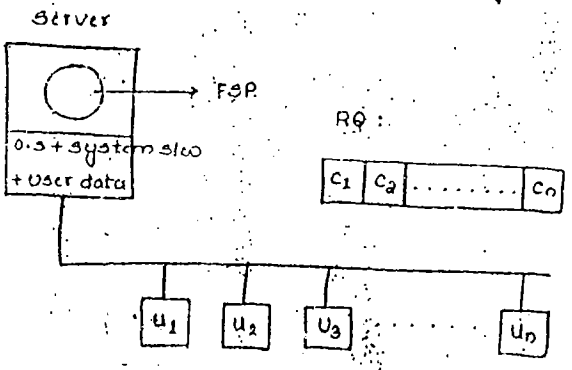
we have booting

- Local booting (done on it's own)
- Remote booting (done through server)

* If server is alive, it transfers the kernel code to the client to get booted, which is called as "remote booting".



Drawback: delayed (i.e., latency).



* Concurrency is achieved through multi-process mechanism (using fork())
* The same code of server (parent) get copied in the childs (clients), i.e., duplication of code is done at all the clients.

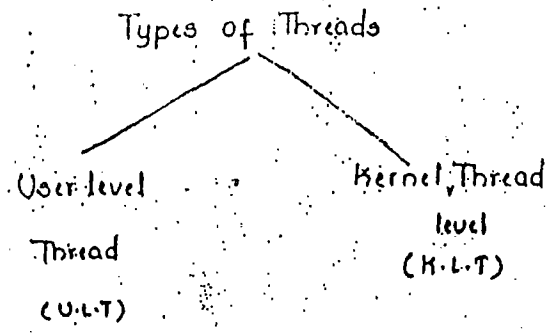
Drawback:

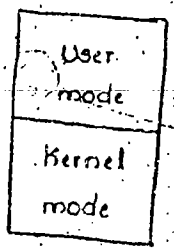
* wastage of resources:
  * Since code is duplicated (i.e., 'n' copies of code, data, files and access -ed by all the clients. same functionality is processed by all clients, instead of single copy and get shared by them).

In non-sharing clients, every independent process must have program counter, General purpose registers and stack for each, where as a single copy of these can be accessed by all the clients in sharing.

Benefits :

* Resource sharing
* cost effective : Economical
* Improved performance.
* Achieve parallelism (multi-CPU).
* |TCB| < |PCB|
  * As process have PCB's, Threads also have TCB's (Id, state, priority)
    * Attributes that can't be shared are stored in TCB (i.e., Id) and which are shared are stored in PCB.
* Context-switch becomes fast and less context-switch overhead. So, there is improved performed.
* Since, it satisfies (5) & (6) th conditions, a thread is called as "Light weight process".





* Threads, which have packages, libraries.

Eg: Java threads (It don't require o.s support)

* O.S visualises it as a process.

* It is transparent to the o.s, that it is only viewed as a process, but not as a thread.

* Java Virtual machine (JVM) allocates the time to the threads.

* User level Thread switching (U.L.T.S) is faster than Kernel level Thread switching (K.L.T.S), because for every processing, there is no need to go to Kernel always, everything is managed at user level.

Drawback :-

* when a process requires IIo, the total process gets blocked instead of a single thread, because of transparency.

* To overcome this drawback, we move to Kernel level threads.

* Threading management can be handled by Kernel level.

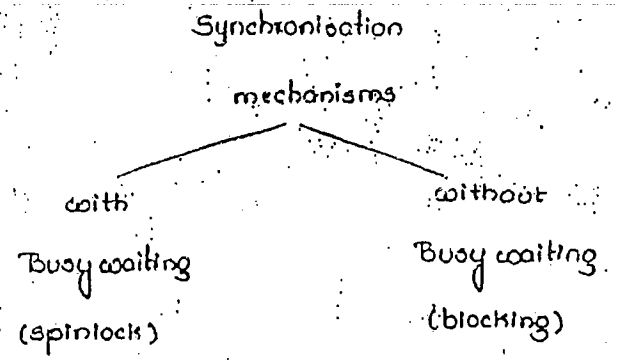
* If one thread requires IIo, then that particular process only gets blocked but not the others.

* context switching also done through Kernel level. so, it is slightly slow compare to user level thread.

Pthread :

* Portable operating system interface for UNIX ⇒ POSIX

Monitors



Spinlock ⇒ busy waiting.

- livelock ⇒ states of process are always ready (or) running.
- Deadlock ⇒ states of process are always blocked

monitors :

It is a collection of procedures, variables and data structures, that are all group together in a special kind of modular package

: Procedures running outside the monitor, cannot access the monitor's internal variables & datastructures. However, they can activate monitor's internal procedures.

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

Producer-consumer problem:

```
monitor   producer.consumer;
begin
      integer count = 0;
      condition full, empty;
      Procedure Enter
      begin
            If (count = n)   wait (full);
            Buffer [in] = itemp;
            in = (in+1) mod n;
            count = count + 1;
            if(count = 1)  signal (empty);  ⇒ // wakeup consumer process
      end
      Procedure Remove
      begin
            if (count = 0)   wait (empty);
            Itemc = Buffer [out];
            out = (out+1) mod n;
            count = count - 1;
            If (count = n-1)   signal (full);
      end
end
Procedure producer
begin
      cobile (True)
      bigin
            Produce_item(Itemp);
            Producer.consumer.enter;
      end
end
Procedure consumer
begin  cobile (True)
       bigin  Producer.consumer.remove;
              process Item(Itemc);
```

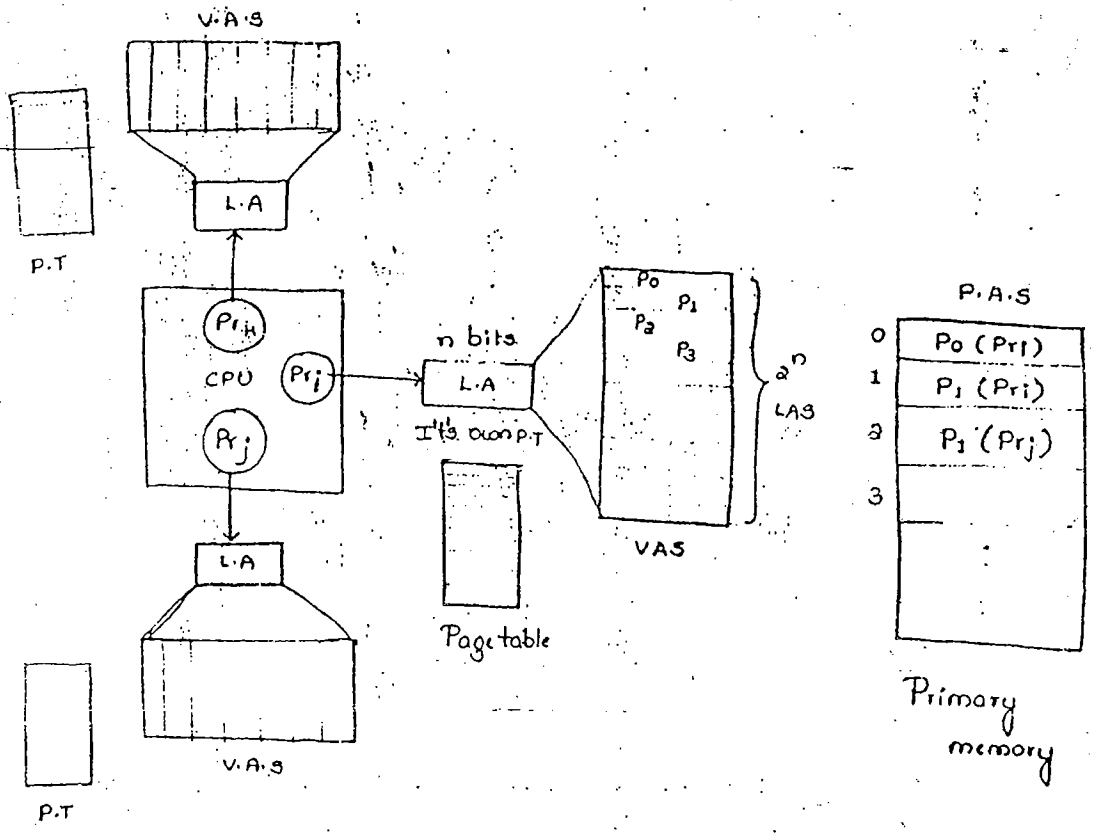
mutex = 1 ⇒ no process is inside

= 0 ⇒ some process is accessing

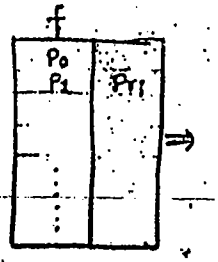
Initially mutex = 1 ;

then DOWN (mutex);

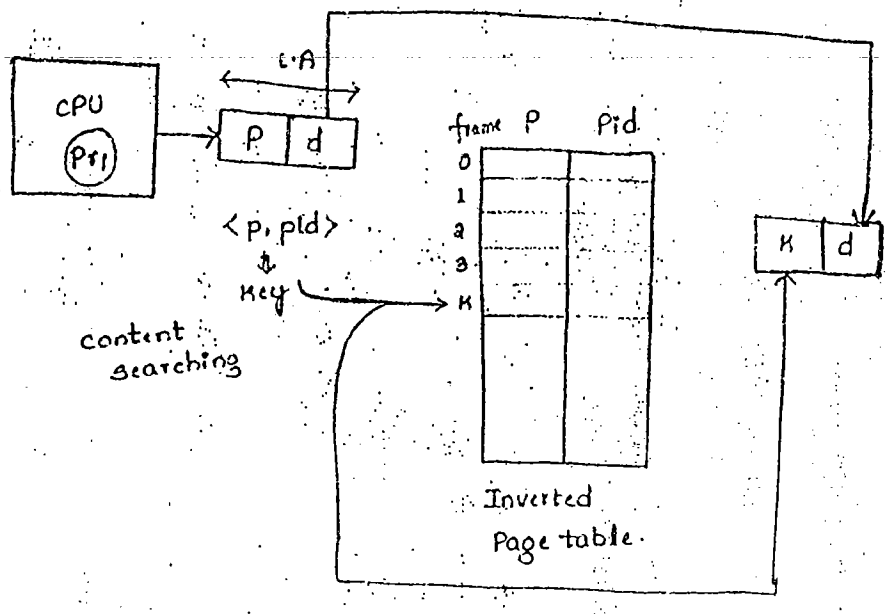
Inverted paging :-



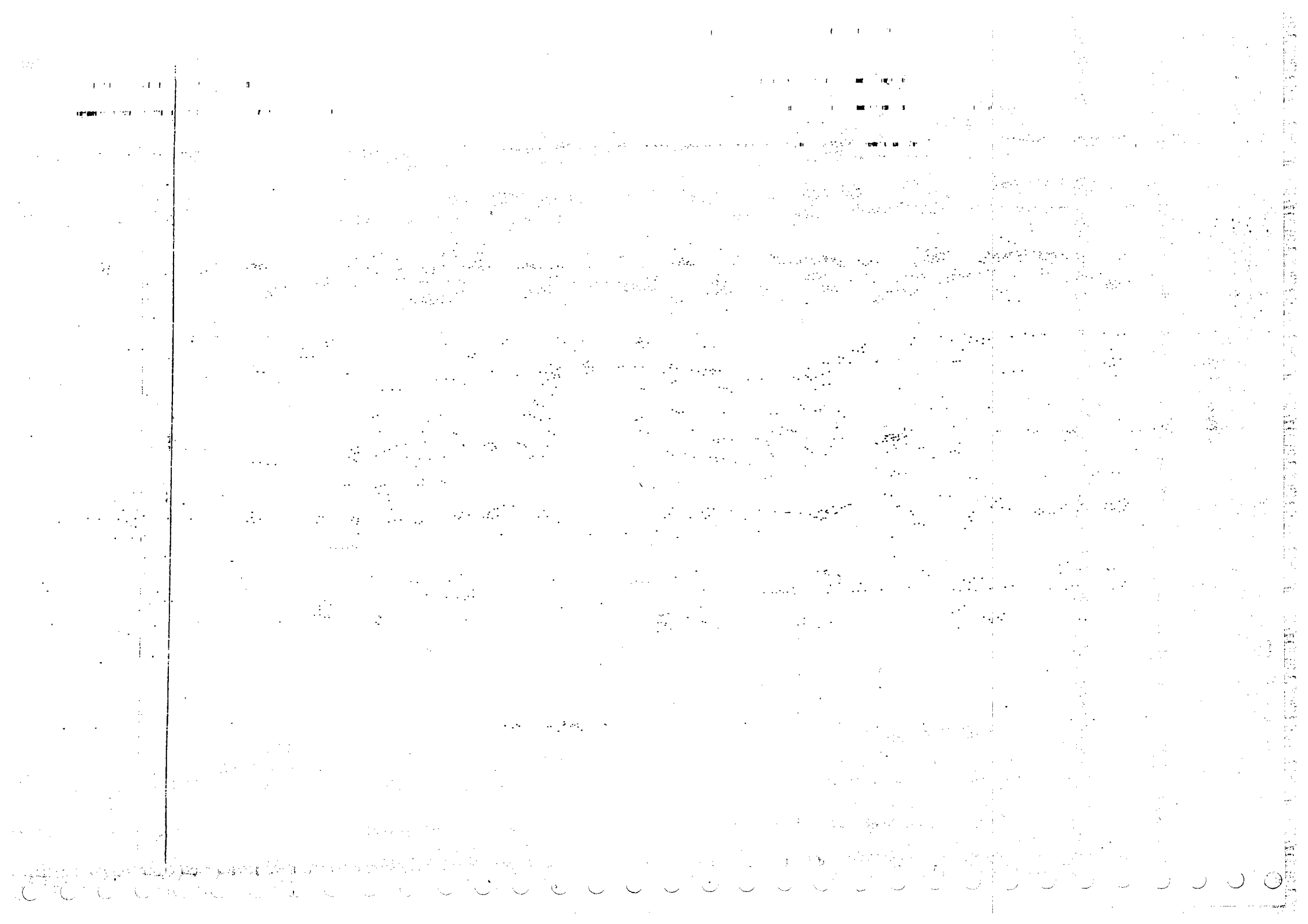
All the processes maintain single primary memory ~~cos~~ within the page table, frames are stored within the page. The Inverted page table contains pages get stored in frames.



⇒ Inverted page Table.



* The Inverted page table maintains the "key" generated by adding <p, pid>, thus, by referring the key, the particular page gets accessed, and the same offset is considered. This reduces the time for searching the content.



08/07/2010

Thursday

Page no : 34

(1) n- processes

context switching — 's' seconds

$T.Q = q$

$n = 5$

CPU | S | P_1 | s | P_2 | s | P_3 | s | P_4 | s | P_5 | s | P_1

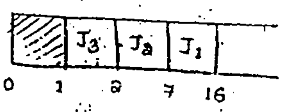
(each P_i slot: q; span P_1 … P_5: t)

$q \leq \frac{t - ns}{n-1}$ (atleast) $q = \frac{t-ns}{n-1}$ (exactly)

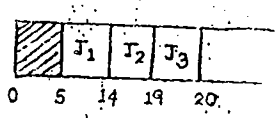
(4)

(a) $\{3,2,1\}$, 1

J	A.T	B.T
1	0.0	9
2	0.6	5
3	1.0	1

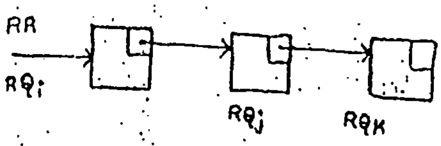


(d) $\{1,2,3\}$, 5



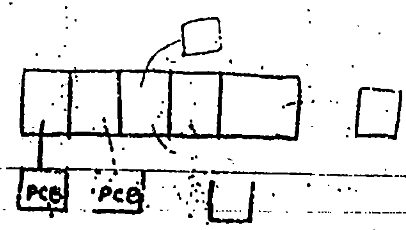
P.no: 42

(1)



priority of process increases

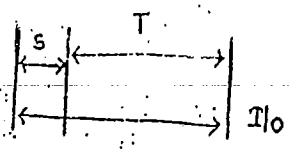
Drawback : starvation.,



(9)

(a) CPU Efficiency (μ) = $\frac{T}{T+S}$ $(TQ = \infty)$

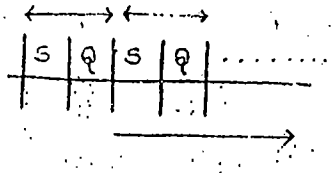
(b) $\mu = \frac{T}{T+S}$ $(TQ > T)$



(c) $\mu = \frac{Q}{Q+S}$ $(S < Q < T)$

(d) $\mu = \frac{Q}{2S}$ $(Q = S)$

$= 50\%$



(e) $\mu = 0$ $(Q = 0)$

(10) Trap & Interrupt :

Trap : Indicates error condition always

Interrupt : Indicates error condition (sometimes but not always)

28/07/2010

Thursday

Rno: 101

(6) $n = 4$, $m = 3$, $(r_1, r_2, r_3) = (9, 3, 8)$

	n	Alloc	Requirt	Available
	P_1	1 1 0	1 0 0	0 0 1
	P_2	1 0 1	0 1 1	safe = $\langle P_3, P_2, P_1, P_4 \rangle$
	P_3	0 1 0	0 0 1	
	P_4	0 1 0	0 2 0	

4/07/2010

Saturday

P.no: 140

5) min. size of partition = max. path length (in sizes of modules)

All the paths are considered as per their sizes and the maximum one is represented.

P.no. 148

(20) e ⇒ frame no. PAS = 64 MB

VAS = 32 bit

PS = 4 KB

P.T size = N * e

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

$$M = \frac{64MB}{4KB} = 2^{14} \text{ frames}$$

$$m = \frac{64MB}{4KB} = \frac{2^{26}}{2^{12}} = 2^{14}$$

e = N * e

= 1M * 2B

= 2MB

P.no: 153

(17) LAS = 8KW (8 pages of 1024 words)

m/r size = 32K (32 frames of 1024 words)

LAS = 8 * 1024

= 8 KW = 13 bits

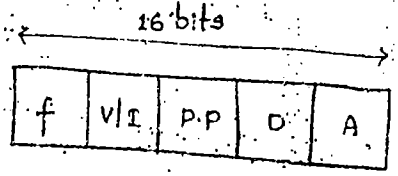
PAS = 32 * 1024 = 32K = 15 bits

30/07/2010

Friday

p.no: 150

(1) (b) LAS = PAS = 2^{16} B



p.t entry

$f = \frac{2^{16}}{2^9} = 2^7$ page size = 512 = 2^9

p.no: 155

(26) (a) Stack :

Since, only top of the stack is accessed, so there is a less no. of page faults.

∴ It is good.

(b) Hashed symbol table :-

It distributes identifiers across the table. so, it generates more no. of page faults.

(c) Sequential search & Binary search :-

likely to cause more page faults than seq. search.

(e) pure code :-

Read only code (non-modified code).

∴ It is good to have non-modified in demand paged environment.

(f) Vector operations :-

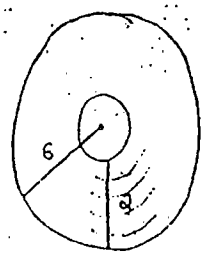
Array (contiguous allocation) ⇒ good.

(g) Indirection :-

Linked lists (non-contiguous allocation) } ⇒ bad.

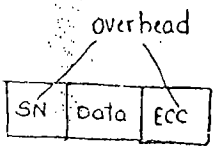
Q: 183.

(8)



no. of tracks = $\frac{(6-2)\,cm}{0.01\,cm}$

= 400 tracks on one surface.



$400 * 96 = 4098 = 4KB$ Each track has 20 sectors

$(400 * 20 * 4KB) * 8$ surfaces

$\Rightarrow 2^9 \times 2^5 \times 2^{12} \times 2^3 = 2^{29} = 512\,MB$

$\simeq 500\,MB$

(b) $R = \frac{60}{3600}$ Sec

$\frac{60}{3600}$ S — 20 * 4 KB

1 Sec — ?

$\frac{3600 * 20 * 4KB}{60}$ B.

P.no: 184

(4) program size = 64 KB

T.S = 30 KB

R = 20 ms

S.T = 30 ms

P.S = 2 KB

20 ms — 32 KB

? — 2 KB

no. of pages = $\frac{64\text{KB}}{2\text{KB}}$ = 32 KB

(a) Time for loading one page = $30\text{ms} + 10\text{ms} + \frac{20 * 2}{32}$ ms

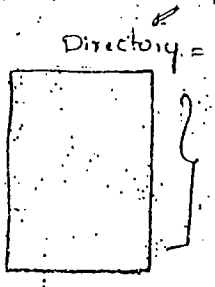
(b) page stored on track to go to that track, we have seek time

$= 30\text{ms} + 10\text{ms} + \frac{20 * 4}{32}\text{ms} * 16$

04/08/2010

Wednesday

(2) (a) max. no. of files ⇒ depends on directory



$\therefore \frac{4\text{KB}}{4\text{B}} = 1\text{K}$

(b) DBA = 8 bits

∴ 256 blocks possible for a file of 8 bits

But, there are 2 data blocks. so, = 256 - 2 = 254.

(254 * 4 KB) ≃ 1 MB

P.no: 183

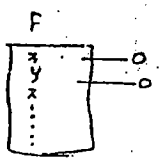
(1) B - blocks
S - DBS
D - DBA
F - free blocks

DBA = D bits
no. of bits = B
free = f

Bit map space consumption is 'B' bits (depends on blocks)

Size of free list (space) = F × D

$\therefore$ $FD < B$



S - DBS
B - blocks
D - DBA
f - free blocks

Disk size = B.S

$2^D \geq B$ (maximum possible)

$(B.S \leq 2^D.S)$

20mB 64mB

Block = 2OK
DS = 20mB
DBA = 16 bits
DBS = 1KB

$2^{16} = 64KB * K = 64MB$ (max.)

2no: 179

(1) b
(2) b (since, it takes more access time, if placed anywhere).
(3) a
(4) b
(5) If there is only one process, then there is no matter whether to use any disk scheduling approach.

so $\Rightarrow$ 0%.

(6) 2 block

In use : 0

free : 0 ⇒ reduces disk size

b) not so serious error but just

6 block

In use : 1

free : 1 c) Serious error.

Since it is free, it may also given to other file, but a file is already being accessing. So, it is considered as a serious error.

To solve this error :-

copy the In-use file to some other block and use it.

P.no: 37

(25) (a)

(26) (a)

P. no : 152

(8) VA = 48 bits

PA = 32 bits

no. of pages = 8K = 2^{13}

no: of entries = $\frac{2^{48}}{2^{13}} = 2^{35} = 32G$ entries

Inverted page table :

$= \frac{2^{32}}{2^{13}} = 2^{19} = 512K$ entries