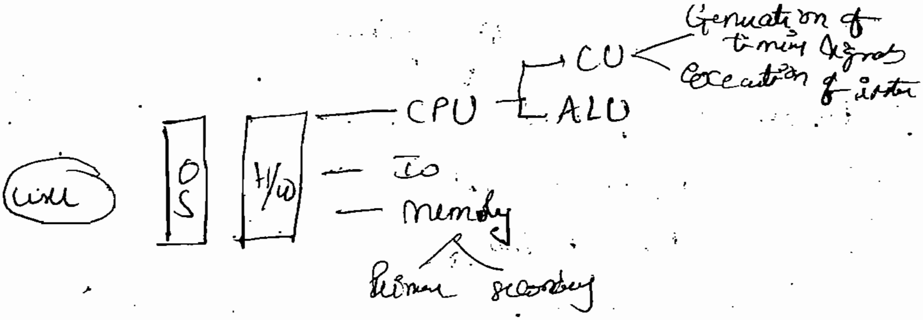
# Operating systems:

- Process management
  - Process concept
  - CPU scheduling
  - Synchronization and Concurrency
  - Deadlocks
  - Threads
- memory management
  - Concepts
  - Techniques
  - virtual memory.
- File system and device management
  - Interfaces
  - Implementation issues
- Protection mechanisms

→ what is an OS?

- Interface b/w user and H/w.
- Control program
- Resource manager (allocator)
- Set of utilities to simplify application development



Resources

H/W

Registers, ALU,
CPU, mem

S/W

Files, semaphores, signals, pipes
msgs.

Goals of an OS:

- user friendly [Convenience)
- Reliable
- Efficiency (in utilization of resources
- Portability (Shd be able to load and use across different environments)

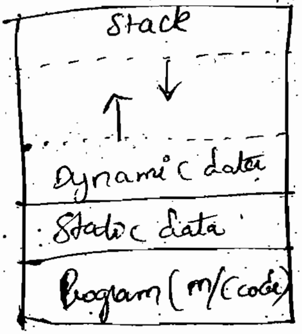
-> Scalability (ability to evolve

## Functions of OS:-

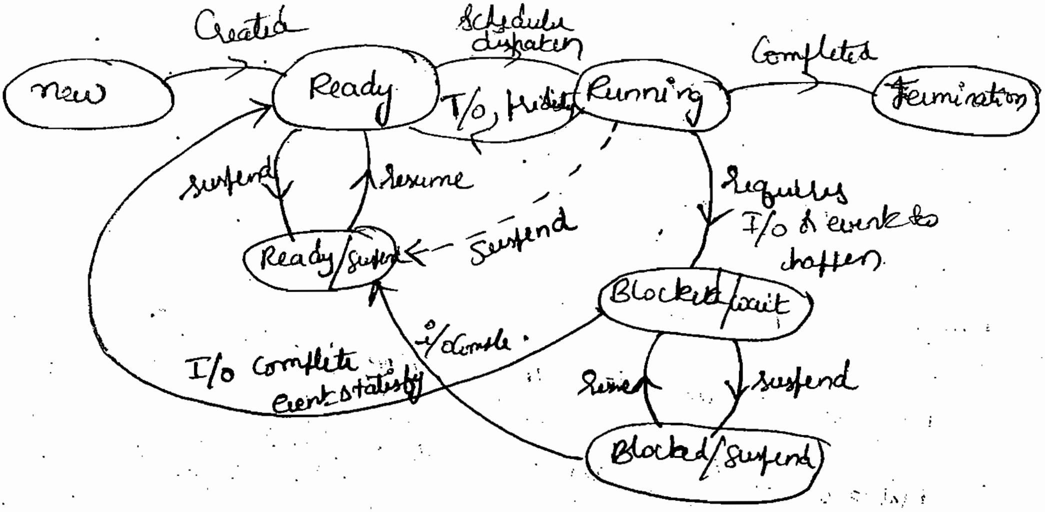
- → Security
- → CPU scheduling
- – mem management
- – Deadlock handling.

## Process Concept:-

- – Program under execution (either in CPU & I/O mem).
- → Programs not under execution (not using CPU) are also called processes, when they are waiting in memory (RAM) for CPU, i.e. using ~~an~~ program is using a resource.
- → The resources allocated to the process are CPU time, memory, files and I/O devices.



## P.S.D. Process state transition diagram:



## Process control block :-

Each process is represented in the OS by a PCB also called task control block.

| | |
|---|---|
| Pointer | Process state |
| Process number | |
| Program Counter | |
| Registers | |
| memory limits | |
| list of open files | |
| | |

PCB Contains many fields of information associated with a specific process, including the following -

Process state: The state may be new, ready, running etc.

Program counter: The address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type, depending on the computer architecture. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

CPU-scheduling information:- This information includes a process priority, pointer to scheduling queues, and any other scheduling parameters.

memory-management information- this includes value of base and limit registers, page tables, or segment tables depending on the memory system used by the O/S.

Accounting information: This i/f includes amount of CPU and real time used, time limits, account numbers, job & process numbers and so on.

I/O status information:- This i/f includes the list of I/O devices allocated to this process, a list of open files and so on.

→ Consider a Computer with n - CPU's and m processes. (n≥1 & m>n) what is the lower limit and UL on the no of processes which can be in the Ready state Running state and blocked state.

| | lower bound | upper bound |
|---|---|---|
| Ready | 0 | m |
| Running | 0 | n |
| Blocked | 0 | m |

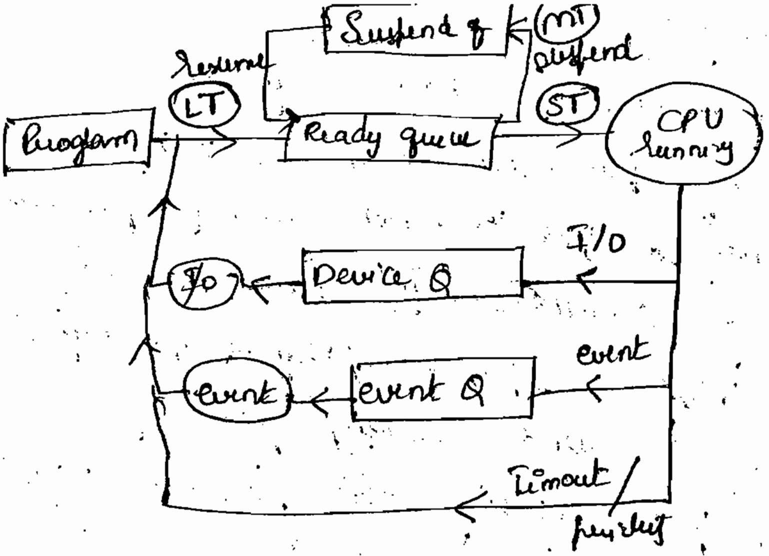
## Multiprogramming :-

A uniprocessor system can have only one ~~So~~ fixo running process at all times. But, to make ~~give~~ the user a feel ~~as~~ that many processes are running simultaneously, the processor switches b/w the processes very frequently. This concept is called multiprogramming. The process of switching the processor b/w processes is called time-sharing.

## multiprocessing :-

A multiprocessor system can have more than one process running at ~~all~~ ~~times~~ given time. The process of running many processes simultaneously is called multiprocessing.

**Degree of multiprogramming :-** The number of processes in the memory is called degree of multiprogramming.



Queuing diagram representation of process scheduling.

Long-term scheduler :- The long-term scheduler or job scheduler, selects processes from secondary memory (disk) and ~~scheduler~~ loads them into memory for execution.

Short-term scheduler or CPU scheduler : selects among the processes that are ready to execute and allocates CPU to one of them

MT scheduler :- Removes processes from memory and they reduces degree of multiprogramming. At some ~~time~~ later time, the process will be reintroduced into memory and its execution can be continued where it left off. This is called swapping.

Processes are generally of two types:

a) I/O-bound process: It spends more of its time doing I/O than it spends doing computations.

b) CPU-bound process: ~~A~~ It generates I/O requests infrequently, using more of its time doing I/O

Context

→ The ~~long~~-term scheduler should select a good process mix of I/O-bound and CPU-bound processes

→ If all processes are I/O bound, then CPU will be idle most of time and ready queue will be almost empty which leaves short term scheduler with little work to do.

→ If all processes are CPU bound, then I/O waiting queues will be almost empty, devices will go unused. End

## Context switching :-

Switching the CPU to another process & requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch. The context of a process can be found in PCB.

## Dispatcher :-

It is a slave of short term scheduler. when a S.T scheduler, decides to ~~schedule~~ move a process from running to ready and another process from ready to running, context switching is needed. All the code needed for this process will be in a module called dispatcher.

The time spent on this process of context switching is called switching time & Dispatch latency.

It is always desirable to reduce CST as much as possible.

Context switching time $\propto$ Contents of size of PCB.

## CPU scheduling techniques & short term scheduler & CPU scheduler

Functions of CPU scheduler :-

making a decision on which process to schedule next.

Goals of CPU scheduler :

- → maximize CPU utilization
- → Be fair to all processes (give chance to all)
- → increase the throughput (number of jobs & processes completed per unit of time).
- → minimize waiting time of processes, TAT, and Response time.

Process times : terms

Arrival time (AT) : AT & submission time is the point of time ~~instance~~ when the process is submitted.

**Burst time (BT)**: BT or service time is the period PU time needed by a process to complete its execution.

**waiting time (WT)**: ~~Times spent by~~ The period of time spent by the process waiting ~~on~~ for CPU in the ready queue is called waiting time (WT).

**Completion time**:-

The ~~tt~~ point of time when the process Completes its execution is called completion time.

**Turn around time (TAT)**: The total ~~time~~ period of time taken by a process from its ~~att~~ arrival till completion is called turn around time.

**Deadline**: The point of time by which the process need to complete is called its deadline.

of <u>Response time</u>: The period of time from the point of submission of a request to the point at which we get first response is called response time.

→ let us assume we have 'n' processes $p_1, p_2, \ldots, p_n$.

→ let AT of $P_i \rightarrow A_i$

BT of $P_i \rightarrow B_i$

CT of $P_i \rightarrow C_i$

Deadline of $P_i \rightarrow D_i$

---

$$\text{TAT}(P_i) = C_i - A_i$$

$$\text{weighted TAT}(P_i) = \frac{C_i - A_i}{B_i}$$

$$\text{Avg TAT} = \frac{1}{n}\sum_{i=1}^{n} C_i - A_i$$

$$\text{Avg wt TAT} = \frac{1}{n}\sum_{i=1}^{n} \frac{C_i - A_i}{B_i}$$

$$\text{weighting time}(P_i) = \text{TAT} - B_i$$

$$= C_i - A_i - B_i$$

Schedule length $(L) = \max(C_i) - \min(A_i)$
(Time to Complete all n processes).

→ CPU scheduling techniques :

These techniques Can be classified as

(i) Preemptive: If a process is deallocated a resource ~~before~~ forcibly before its completes ~~its~~ the resource usage, ~~is~~ then such a technique is called preemptive technique. In scheduling context, the resource is CPU and techniques are scheduling techniques.

(ii) Non Preemptive: If a process is allowed to ~~use~~ ~~to~~ release an allocated resource only after it finishes ~~its~~ ~~usage~~ the resource usage, then such technique is called non preemptive technique.

→ If there are 'n' processes, then how ~~possible~~ schedules are possible with a non preemptive scheduling technique.

$n!$

→ Throughput ($\mu$) = $\frac{\text{no of processes}}{\text{total time}}$.

→ Deadline over run ($P_i$) = $C_i - D_i$

$C_i - D_i$ = +ve − Crossed deadline
= −ve − before time
= 0 − on time

$P_i$. $C_i$

FCFS: It is based on AT, It is non-preemptive technique.

| P.NO | AT | BT |
|---|---|---|
| 1 | 0 | 4 |
| 2 | 1 | 5 |
| 3 | 2 | 6 |
| 4 | 3 | 7 |
| 5 | 4 | 3 |

| 4 | 5 | 6 | 7 | 3 |
|---|---|---|---|---|

0 4 9 15 22 25

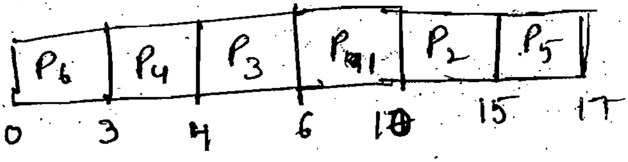
| P.NO | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| 1 | 0 | 4 | 4 | 4 | 0 |
| 2 | 1 | 5 | 9 | 8 | 3 |
| 3 | 2 | 6 | 15 | 13 | 7 |
| 4 | 3 | 7 | 22 | 19 | 12 |
| 5 | 4 | 3 | 25 | 21 | 18 |

$L = 25$, $\mu = \frac{5}{25} = 1/5$

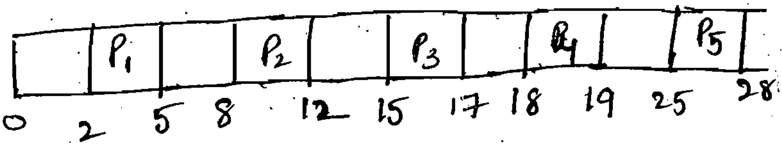
Avg TAT = $65/5 = 13$ Avg WT = $\frac{40}{5} = 8$.

Ex

| Pid | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| 1 | 4 | 4 | | | |
| 2 | 5 | 5 | | | |
| 3 | 2 | 2 | | | |
| 4 | 1 | 1 | | | |
| 5 | 7 | 2 | | | |
| 6 | 0 | 3 | | | |



| Pid | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 3 | 0 |
| 2 | 8 | 4 | 12 | 4 | 0 |
| 3 | 15 | 2 | 17 | 2 | 0 |
| 4 | 18 | 1 | 19 | 1 | 0 |
| 5 | 25 | 3 | 28 | 3 | 0 |



$L = 28$

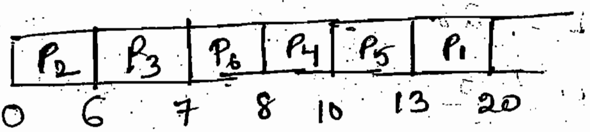
$\mu = \frac{5}{28}$

There is a Convoy effect, as all the other processes wait for the one big process to get off the CPU. To overcome this, we go for SJF & SJN.

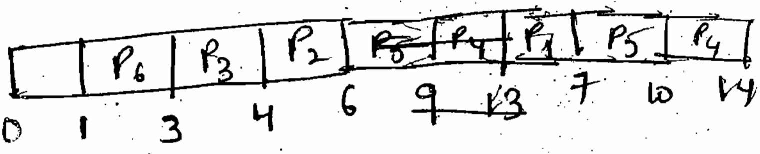
# Shortest Job First (SJF) & Shortest-process Next (SPN)

This is based on BT. ~~mode: preemptive.~~
mode: non preemptive.

| PNO | AT | BT |
|---|---|---|
| 1 | 0 | 7 |
| 2 | 0 | 6 |
| 3 | 2 | 1 |
| 4 | 3 | 2 |
| 5 | 4 | 3 |
| 6 | 5 | 1 |



| PNO | AT | BT |
|---|---|---|
| 1 | 5 | 1 |
| ✓2 | 4 | 2 |
| ✓3 | 3 | 1 |
| 4 | 1 | 4 |
| ✓5 | 2 | 3 |
| ✓6 | 1 | 2 |



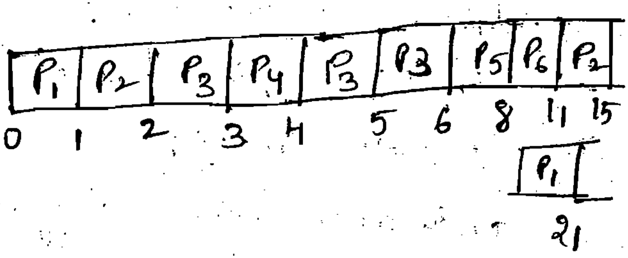
# Shortest Remaining time first (SRTF)

It is preemptive version of SJF.

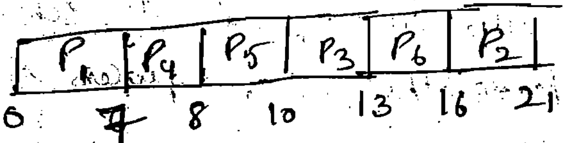
Criteria : BT , mode : P.

Ex:

| PNO | AT | BT |
|---|---|---|
| 1 | 0 | ~~7~~ 6 |
| 2 | 1 | ~~5~~ 4 |
| ✓ 3 | 2 | ~~3~~ ~~2~~ 1 |
| ✓ 4 | 3 | ~~1~~ 0 |
| ✓ 5 | 4 | 2 |
| ✓ 6 | 5 | 3 |

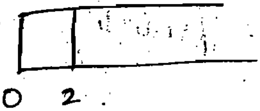


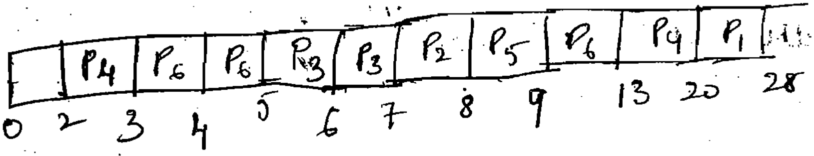
NP-SJF



→

| PNO | AT | BT |
|---|---|---|
| ✓ 1 | 4 | 8 |
| ✗ 2 | 6 | ~~1~~ 0 |
| ✗ 3 | 5 | ~~2~~ ~~1~~ 0 |
| ✗ 4 | 2 | ~~8~~ 7 |
| ✗ 5 | 7 | 1 |
| ✗ 6 | 3 | ~~6~~ ~~5~~ 4 |





→ SJF (Pre and non prem) favours shorter jobs and causes starvation to longer jobs.

→ It gives high throughput at any point of time compared to ~~P~~ any scheduling technique ~~So, it is optimal scheduling algorithm.~~

→ ~~It this technique is~~ and minimum average waiting time for a given set of processes.

→ Though SJF is optimal, it cannot be implemented at the level of ST CPU scheduling as we cannot know the exact BT of a process ~~exactly~~ before running it.

→ One approach is to approximate SJF scheduling, by predicting the burst time of a process.

→ We expect that the ~~next~~ CPU burst of the next process will be similar in length to the CPU burst of the previous ones.

→ Let $t_n$ be the length of the $n^{th}$ CPU burst and let $\tau_{n+1}$ be our predicted value for the next CPU burst. Then, for $\alpha$, $0 \le \alpha \le 1$, define

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

→ This formula defines an exponential average.

→ $t_n$ contains most recent information and $\tau_n$ stores the past history.

→ The parameter $\alpha$ controls the relative weight of recent and past history in our prediction.

→ If $\alpha = 0$, $\tau_{n+1} = \tau_n$ and so recent information has no effect.

→ If $\alpha = 1$, then $\tau_{n+1} = t_n$ and only the most recent CPU burst matters.

→ If $\alpha = 1/2$, then $\tau_{n+1} = \frac{1}{2}(t_n + \tau_n)$, i.e equal weightage is given to both histories

# Priority scheduling: (Based on priority).

-> The SJF algo is a special case of the general priority-scheduling algorithm.

P/S Priority

Static -> priority won't change

Dynamic -> Priority can be changed in between.

-> It can operate in preemptive & non preemptive mode.

~~NP priority: or~~

NP - priority scheduling:

~~It codes exact~~ out of all ~~arrived pro~~ processes submitted the technique will select a process with highest priority and executes it first.

-> highest number may indicate highest priority ~~&~~
lowest number may indicate highest priority.

-> here let us assume highest number ~~also~~ ~~[illegible]~~ indicates highest priority.

| Priority | P.NO | AT | BT |
|---|---|---|---|
| 4 | 1 | 0 | 4 |
| 5 | 2 | 1 | 5 |
| 6 | 3 | 2 | 9 |
| 3 | 4 | 4 | 2 |
| 7 | 5 | 5 | 7 |

| | | | | | |
|---|---|---|---|---|---|
| P1 | P3 | P5 | P2 | P4 | |

0 4 13 20 25 27.

| Priority | P.NO | AT | BT |
|---|---|---|---|
| ✓ 4 | 1 | 6 | 4 |
| ✓ 3 | 2 | 10 | 5 |
| ✓ 2 | 3 | 8 | 6 |
| ✓ 2 | 4 | 4 | 2 |
| ✓ 1 | 5 | 2 | 3 |
| 1 | 6 | 12 | 1 |

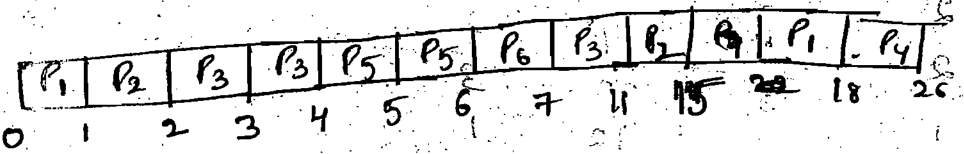
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | P5 | P4 | P1 | P2 | P3 | P6 | |

0 2 5 7 11 16 22 23.

## b) Preemptive - priority:

This technique continues execution of a process till the arrival of next process, and, if the next process has higher priority, then preempt the current process.

| | Pri | P.NO | AT | BT |
|---|---|---|---|---|
| ✗ | 4 | 1 | 0 | ~~4~~ 3 |
| ✗ | 5 | 2 | 1 | ~~5~~ 4 |
| ✗ | 6 | 3 | 2 | ~~6~~ 4 |
| ✓ | 2 | 4 | 3 | 8 |
| ✗ | 8 | 5 | 4 | ~~2~~ 1 |
| ✗ | 7 | 6 | 5 | 1 |



## Round-Robin Scheduling:-

The RR scheduling algo is designed especially for time sharing systems.

» It is useful for multiprogrammed, multiuser and interactive systems like unix.

→ It is similar to FCFS scheduling, but preemption is added.

→ Criteria – AT + time quantum

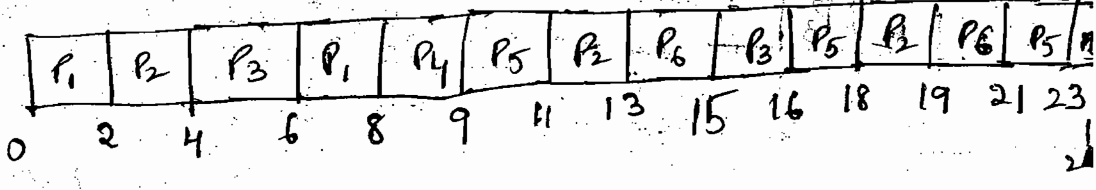
→ ~~A small / cpu~~

→ Time quantum (or time slice is small unit of time.

→ To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.

| PNO | AT | BT |
|---|---|---|
| 1 | 0 | ~~4~~ ~~2~~ 0 |
| 2 | 1 | ~~5~~ ~~3~~ 1 |
| 3 | 2 | ~~3~~ 1 |
| 4 | 3 | 1 |
| 5 | 4 | ~~6~~ ~~4~~ 2 |
| 6 | 5 | ~~5~~ ~~3~~ 1 |

~~$P_1$~~ ~~$P_2$~~ ~~$P_3$~~ ~~$P_1$~~ ~~$P_4$~~ ~~$P_5$~~ ~~$P_2$~~ ~~$P_6$~~ ~~$P_3$~~
~~$P_5$~~ ~~$P_2$~~ ~~$P_6$~~ ~~$P_5$~~ $P_6$



->

| PNO | AT | BT |
|---|---|---|
| 1 | 1 | ~~4~~ ~~1~~ 0 |
| 2 | 3 | ~~3~~ 0 |
| 3 | 5 | ~~4~~ ~~1~~ 0 |
| 4 | 6 | ~~5~~ ~~2~~ 0 |
| 5 | 8 | ~~6~~ ~~3~~ 0 |
| 6 | 9 | ~~7~~ 4 |

TQ = 3

~~$P_1$~~ ~~$P_2$~~ ~~$P_1$~~ ~~$P_3$~~ ~~$P_4$~~ ~~$P_5$~~ ~~$P_6$~~ ~~$P_3$~~ ~~$P_4$~~ ~~$P_5$~~ $P_6$

| $P_1$ | $P_2$ | $P_1$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
|---|---|---|---|---|---|---|---|---|---|---|

1 4 7 8 11 14 17 20 21 23 26 30

->

| P.NO | AT | BT |
|---|---|---|
| 1 | 6 | 4 |
| 2 | 2 | ~~5~~ 3 |
| 3 | 3 | ~~6~~ 4 |
| 4 | 1 | ~~8~~ 6 |
| 5 | 4 | 10 |
| 6 | 5 | 3 |

TQ = 2

o/p

~~$P_4$~~ ~~$P_2$~~ $P_3$ $P_4$ $P_5$ $P_6$ $P_2$

| $P_4$ | $P_2$ | $P_3$ |
|---|---|---|

1 3 5

# Highest Response ratio next : (HRRN)

~~At Ex~~ Response ratio = $\frac{w+S}{S}$ .

→ The scheduling is based on response ratio.

→ It is a non-preemptive scheduling technique.

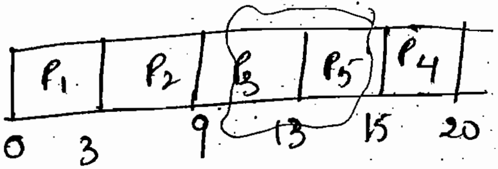
→ Compute the RR of each remaining process after execution of every process.

Example

| PNO | AT | BT | WT |
|---|---|---|---|
| 1 | 0 | 3 | |
| 2 | 2 | | |
| 3 | 4 | | 5 |
| | 6 | 5 | 3 |
| 5 | 8 | | |

SJF algo

HRRN



## Multilevel queue scheduling :

→ Processes can be classified into different groups.

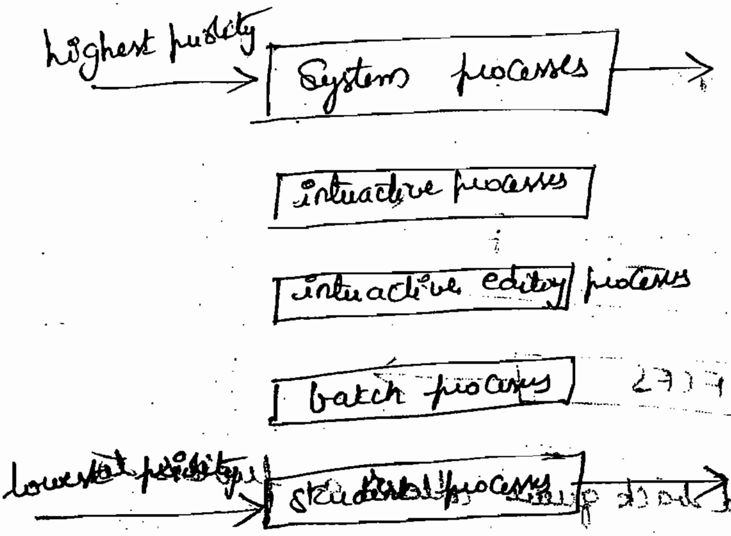
→ A common way to classify processes is division is made b/w foreground (or interactive) processes and background (or batch) processes.

→ A multilevel queue-scheduling algorithm partitions the ready queue into several seperate queues.

→ The processes are permanently assigned to one queue, generally based on some property of the process, such as, memory size, process priority, or process type.

→ Each queue has its own scheduling algorithms. For example, foreground and back queue might be scheduled by an RR algo, while the background queue is scheduled by an FCFS algo.

→ In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

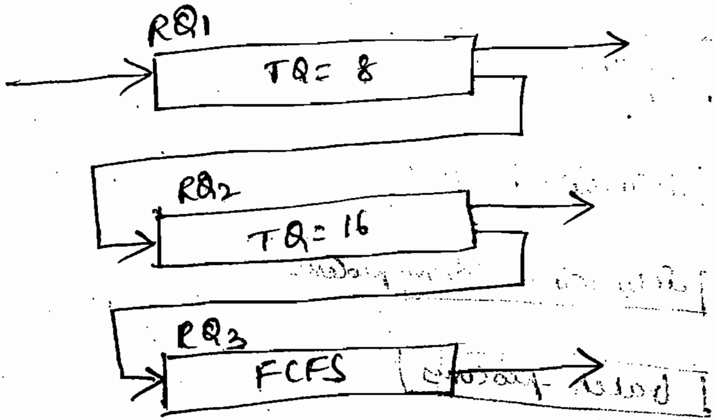


Advantage: when process comes, it is joined to q of its type and different scheduling techniques can be operated in different q's.

Disadvantage:
If ~~upper~~ higher priority q is empty, then only lower priority q can be operated.
So processes in lower priority q suffer from starvation.

→ To over Come starvation, feedback 'q' is used.

multilevel feedback Q scheduling :-



→ multilevel feedback queue allows a [illegible] move between queues.

→ First, a process enters RQ1, when its [illegible] [illegible] execute, it will execute [illegible] and then it will be sent to RQ2, if it needs some more CPU time to finish its execution.

→ once RQ1 gets empty, then RQ2 will be ~~selected~~ scheduled for execution. Here a process will get a ~~RQ of~~ CPU time of 16 units ~~to get~~.

and thus it will be sent ~~back to~~ RQ3, if it still needs more time. ~~Here~~

→ In RQ3, a process will get executed till it completes.

→ The number of queues used and the scheduling algorithm used within each queue are system dependent.

→ This scheduling algo gives highest priority to any process with CPU burst of 8 milliseconds or less.

→ Next priority will be to process with 8-24 milliseconds

→ long processes automatically sink to RQ3 and gets served in FCFS order.

(schedules threads)

→ Solaris 2, uses multilevel queueing scheduling containing the following queues

(i) Real time

(ii) system

(iii) time sharing and interactive

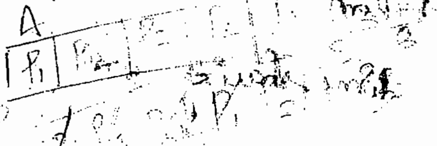
→ window 2000 uses ~~p~~ preemptive priority scheduling. Here it uses priority number from 0 - 31.

Real time process will be give priorities 16-31 and other 1-15.

→ Priority '0' is given to memory management thread. this will execute when no process is running.

→ Linux ~~priorities~~ 8 types / separate sch. uses RR ~~[illegible]~~ interactive processes and ~~priority~~ queue ~~for~~ Real ~~time~~ process.

Ex: Consider a system ~~integration~~ processes ~~arriving in~~ arriving in same order at t=0. (A B C) Their BTs are (1, 8, 1) respectively. ~~calculate~~ What is the completion time of A under RR scheduling, with time slice of 1 unit.



9

# Process synchronization :

## IPC and process synchronization:-

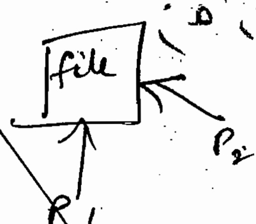
→ The Concurrent processes executing in the O.S. may be either independent or cooperating processes.

→ A process is independent if it cannot affect or be affected by the other processes executing in the system.

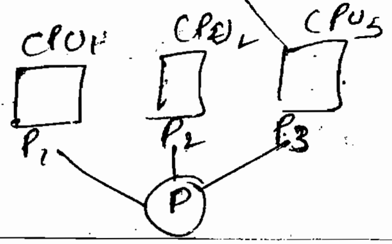
→ A process is cooperating if it can affect and be affected by the other processes executing in the system.

The reasons for using cooperating processes are:

(i) Information sharing:



(ii) Computation speedup:



modularity! for many tasks

Convenience:

→ These Cooperating processes Can Communicate with each other via interprocess communication facilities like shared memory and message passing.

## Race Conditions:

Let 'a' is a shared variable

a++;

↓ compiler

1 mov a, reg1;

2 Add reg1, 1;

3 move reg1, a;

$P_2$ a--

1 move a, Reg1;

~~2 Add~~

2. Sub reg1, 1;

3. move reg1, a;

reg: 4

$P_1$ : 1, 2 preempt

: 1, 2

$P_2$

|  |  |  |  |
|---|---|---|---|
| $P_1$ : 1, 2 | $P_2$ : 1, 2 | $P_1$ : 3 | $P_2$ : 3 |

|  |  |  |  |
|---|---|---|---|
| $P_1$ : 1, 2 | $P_2$ 1, 2 | $P_2$ : 3 | $P_1$ : 3 |

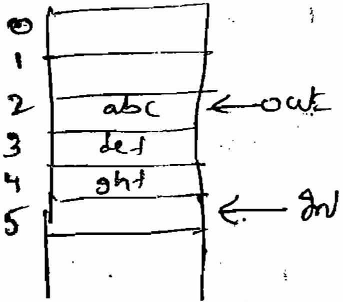
Situations where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

<u>Ex</u>: Print spooler:

→ when a process wants to print a file, it enters the file name in a special spooler directory.

→ Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names.

Spooler directory



$P_1$, $P_2$ both wants to queue a file for printing.

```
x  next_free_slot = in
y  spooler[next_free_slot] = abc
z  { in++;
   }
```

| | | | | | |
|---|---|---|---|---|---|
| $P_1$: x | $P_2$: x | $P_1$: y | $P_2$: y | $P_1$: z | $P_2$: z |

~~Critk~~

Critical region & critical section!

→ The key to ~~avai~~ avoid race conditions is to find some way to prohibit more than one process from reading and writing to shared data.

→ In otherwords, ~~we need~~ what we need is mutual exclusion, that is, some way of making sure that if one process is using ~~bo~~ shared variable or file, the other process ~~[illegible]~~ excluded from ~~[illegible]~~ doing the same.

→ ~~A process will be~~

→ (~~part~~ of the time, a ~~process~~ [illegible] busy doing internal computations ~~[illegible]~~ that do not lead ~~to race~~ ~~conditions~~ ~~and~~ Remaining time,

→ it accesses ~~the~~ shared memory.

→ The part of that program where the ~~[illegible]~~ memory is accessed is called the critical region or critical section.

→ If we could find a solution such that no two processes can ever be in their critical regions at the same time, we could avoid races.

→ Such a solution is said to be a good solution, if it satisfies the following ~~conditions~~: requirements

1) ~~[illegible]~~ No two processes may be simultaneously inside their critical regions (mutual exclusion).

2) No ~~assumptions~~ may be made about ~~speeds~~ or the number of CPUs.

3) ~~No process running~~ outside its critical region may block other processes. ([illegible])

4) No process should have to wait [illegible] to enter its critical region (Bounded waiting).

→ Solutions to avoid race conditions

Disabling interrupts: (H/w solutions)

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable then just before leaving it.

→ with interrupts disabled, no clock interrupts can occur.

→ The CPU is only switched from process to process as a result of clock / other interrupts and with interrupts turned off, the CPU will not be switched to another process.

ME ✓, No assumptions X, Progress ✓

BW X

<u>Bounded waiting</u>: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

<u>Disadvantage</u>: The control over interrupts should never be given to users. (what if an user disables an interrupt and never enables it).

**Lock variables:** (S/W solution), 2 process, BW.

| entry section | ~~if (lock~~ |
|---|---|
| | while (lock == 1); |
| | lock = 1; |
| CS | CS |
| exit section | lock = 0; |

ME X, BWX, no AV, Progress ✓

✓ suspend X

**Strict alternation:** SW, 2 process, BW, ME

```
P0                      P1
{                       {
while(turn!=0);         while(turn!=1);
CS                      CS
turn=1;                 turn=0;
NCS                     NCS
}                       }
```

→ Continuously testing a variable until some value appears is called busy waiting.

→ busy waiting wastes CPU time

ME ~~✗~~, ✓, P X, NO assum ✓, BW X

The TSL instruction:- HW ~~so~~ and SW solutions

~~Test~~ Set Lock

TSL RX, LOCK.

-> ~~It~~ The above Command reads the contents of the memory word 'lock' into the register RX and then stores a nonzero value at the memory address 'lock'.

-> The operations of reading the word and storing into it are guaranteed to be indivisible.

-> No other process can access the memory word ~~until~~ until the instruction is finished.

-> The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

80,000

```
enter-region;
~~Entry~~:
    TSL REGISTER, LOCK // copy lock to register and set
    CMP REGISTER, #0 // was lock zero?
    JNE enter region // if it was nonzero, lock was s
    RET // return to caller; critical region entered.

leave-region:
    MOVE LOCK, #0                0 in lock.
    RET // Return to Caller
```

Peterson's solution

lock variable

It continues

may

and waiting

lock to] ;

so loop;

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

int turn;
int interested [N];
void enter-region (int process) /* process is 0 & 1 */
{1 int other; /* number of the other process */
2 other = 1-process; /* the opposite of process */
3 interested [process] = True; /* show that you
                                   are interested */
4 turn = process; /* set flag */
5 while( turn == process && interested [other] = TRUE);
}
void leave-region (int process) /* process : who is
                                   leaving */
{ interested [process] = FALSE; /* indicate
                                   departure from critical
                                   region */
}
```

-> A process before entering critical section will call enter-region with its own process number, 0 or 1, as parameter.

-> After it has finished with the shared variables, the process calls leave-region to indicate that it is done and to allow the other process to enter, if it so desires.

**Analysis:**

-> Initially neither process is in its critical region. Now process 0 calls enter_region. It indicates its interest by setting its array element and sets turn to '0'. Since process 1 is not interested, enter_region returns immediately.

-> If process 1 now calls enter_region, it will hang there until interested[0] goes to FALSE, an event that only happens when process '0' calls "leave-region" to exit the critical region.

-> Now consider the case that both processes call enter_region almost simultaneously.

$P_1$ :

| Turn |
|---|
| 0 |

Interested

| 0 | 1 |
|---|---|
| True | False |

Both will store to their process number in 'turn'. whichever store is done last is the one that Counts; the first one is overwritten and lost.

$P_0$: 1, 2, 3, 4 | $P_1$: 1, 2, 3, 4 $P_0$ : 5 | $P_0$ : 5 | CS 6 |

$P_1$: 5 : CS : 6 | .

Interested

| 0 | 1 |
|---|---|
| ~~True~~<br>False | True |

| Turn |
|---|
| ~~0~~ 1 |

$P_1$:

Suppose that process 1 stores last, so 'turn' is 1. when both processes come to while statement, process '0' executes it zero times and enters its critical region. Process '1' loops and does not enter its critical region until process 0 exits its critical region.

ME ✓, Progress ✓, BW ✓, no assumptions.

Notes:

Both peterson's and the solution using TSL are correct but both have a defect of ~~is~~ requiring busy waiting and priority inversion problem.

Consider a computer with two processes, 'H', with high priority and 'L', with low priority. The scheduling rules are such that 'H' is run whenever it is in ready state. At a certain moment, L is only in the system and it enters critical region, now 'H' becomes ready to run and enters the ready queue. 'H' now begins busy waiting, but since 'L' is never scheduled while H is running, L never gets the chance to leave its critical region. So H loops forever. This situation is referred to as the priority inversion problem.

## Semaphores

A semaphore 'S' is an integer variable, which is accessed only through two standard atomic operations; "wait" and "signal".

---

→ one of the simplest ways to avoid busy waiting is to use a pair "sleep" and "wakeup".

→ "Sleep" is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awaked.

---

~~In the~~ ("wait" is generalization of ~~to~~ "sleep".

"Signal" is generalization of ~~a~~ "wakeup".

```
Struct semaphore
{   int value;
    ~~Queue Type L~~; struct process *L;
}
```

→ Semaphore is a OS resource Execute in OS Kernel atomically.

Semaphores

Counting semaphores
-∞ to ∞

Binary semaphores
0, 1.

Counting semaphore :

```
Down (struct semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        ~~Block the process and put its PCB in S.L(...)~~; add this pro[illegible] S.L; block();
    }
}
```

```
UP(struct Semaphore S)
{
        S.value = S.value +1;
        if (S.value ≤ 0)
        {
            ~~Select a of blocked process from~~
            ~~S.L() and wakeup();~~
            Remove a process P from S.L;
        }   wakeup(P);
}
```

→ Positive value of 'S' indicates no of successful down operations that can be performed.

→ -ve value indicates no of blocked processes

→ A process may get blocked only while performing down operation.

Ex: In a certain computation, the value counting S is initialized to S=13. then

The following operations were ~~[illegible]~~ ~~perf~~ performed on the semaphore in the given order. What is the status of it after these ...

15 P, 12 V, 10 P, 8 V, 8 P, 7 V, 6 P, 1 V.

P - down V - up.

-2, 10, 0, 8, 0, 7, 1, 2.

→ S = 11

10 P, 4 V, 18 P, 6 V, 8 P, 12 V, 3 P, 1 V.

→ Binary Semaphore (mutex).

~~The~~ In general they are used for mutual exclusion hence called (mutex).

It takes only 0 & 1 ~~of~~

$$\underset{\text{busy}}{0}\ \&\ \underset{\text{free}}{1}$$

Struct B Semaphore
{ Enum value (0, 1);
Queue type L;
}

```
BDown( Struct BSemaphore S)
{   if (S.value ==1)
      {   S.value =0;
          return;
      }
    else
      {   Block the process; and add it
          to the queue;
      }
}
```

→ The no of blocked processes is not known in binary processes semaphores.

**UP:**

→ when a process performs UP operation and if it finds Q not empty, then S=0 only and it wakes up one process.

→ only if no process is waiting blocked, then it makes S to 1.

```
BUP(struct BSEMAPHOE S)
{
    if (S.L() is not empty)
    {  select a process from S.L() and
         wakeup();
    }
    else
        S.value = 1;
}
```

Ex;

| Pi | Pj |
|---|---|
| 1 P(S) | 1 P(T) |
| 2 P(T) | 2 P(S) |
| CS | CS |
| 3 V(T) | 3 V(S) |
| 4 V(S) | 4 V(T). |

~~SCS~~ $\boxed{CR}$

dead lock.

So for no dead lock, both should have same entry section i.e

$P_0$

$P(S)$
$P(T)$

$P_1$

$P(S)$
$P(T)$.

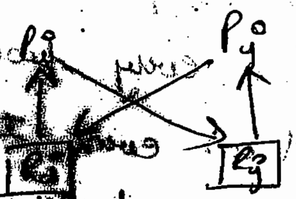
→ we say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

→ Another process related to deadlock is indefinite blocking or starvation, a situation where processes wait indefinitely [illegible] at semaphore

# Deadlocks:

⇒ Permanent blocking of set of processes is called deadlock.

> Primary requirement for set of DL is two or more processes.



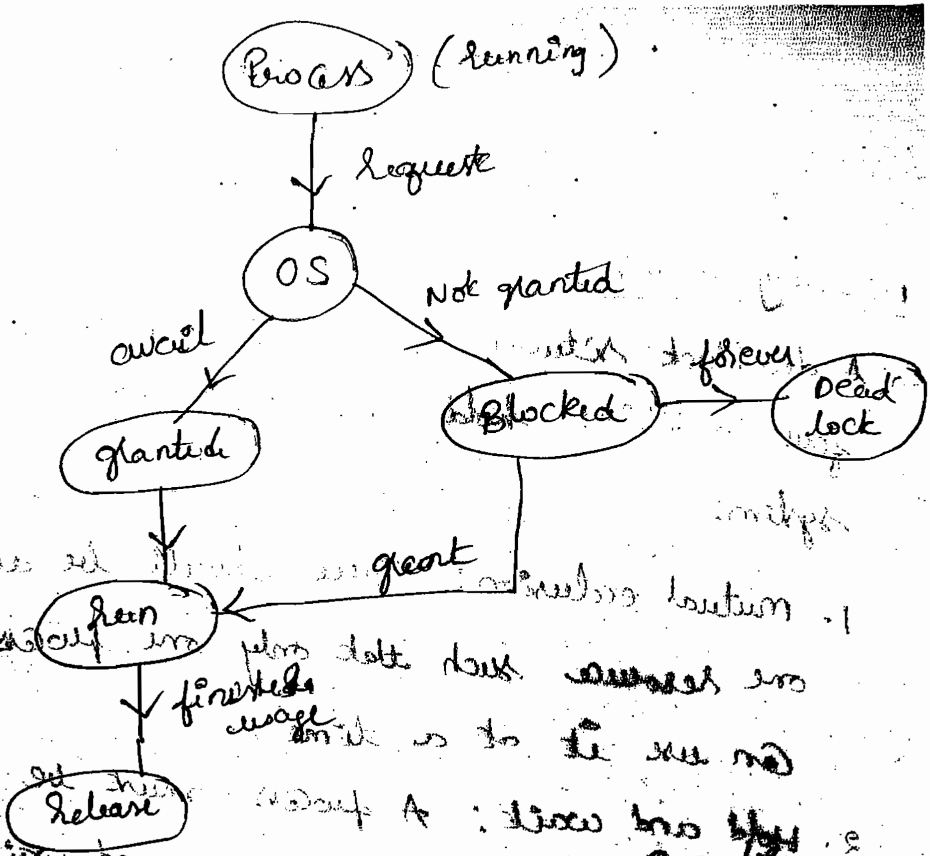
> System model

→ The system consists of finite number of resources to be distributed among a number of competing processes.

→ Each resource may have several identical instances.

CPU - 2 instances.

Printer -5 instances.



→ Request and release can be accomplished through "wait" and 'signal' operations on semaphores.

## Deadlock characterization

- Necessary conditions
- Resource allocation graph

Necessary Conditions:-

→ A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion: There should be atleast one resource such that only one process can use it at a time.

2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; i.e, a resource can be released only voluntarily by a process holding it, after that process has completed its task.

4. Circular wait: A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that $P_0$ is waiting for a resource that is held by $P_1$, $P_1$ is waiting for a resource that is held by $P_2$ .... $P_{n-1}$ is waiting for a resource that is held by $P_n$ and $P_n$ is waiting for a resource that is held by $P_0$.

## Resource-allocation graph:

→ Deadlocks can be described more precisely in terms of directed graph called a system resource allocation graph.

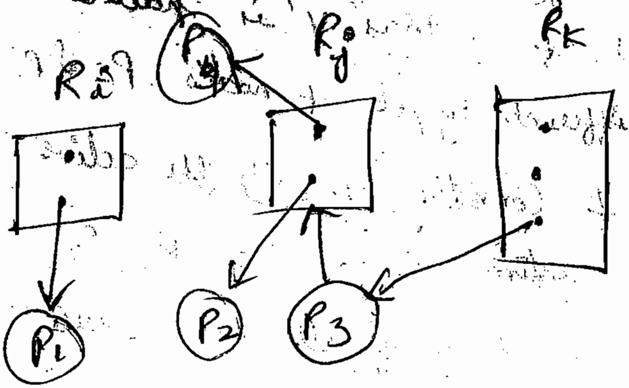
→ The graph consists of a set of vertices V and set of edges E.

→ The set of vertices V is partitioned into two different types of nodes $P = \{P_1, P_2, \ldots P_n\}$ the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \ldots R_m\}$, the set consisting of all the resource types in the systems.

→ A directed edge from process $P_i$ to resource type $R_j$ is denoted by $P_i \rightarrow R_j$; it signifies that process $P_i$ requested an instance of resource type $R_j$ and is currently waiting for that resource.

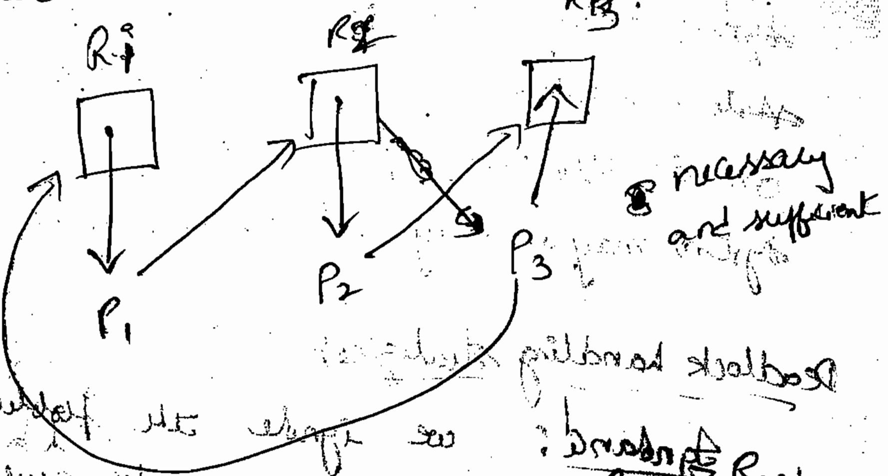
→ A directed edge from resource type $R_j$ to process $P_i$ is denoted $R_j \rightarrow P_i$; it signifies that an instance of resource type $R_j$ has been allocated to process $P_i$.

→ A directed edge $P_i \rightarrow R_j$ is called a request edge, a directed edge $R_j \rightarrow P_i$ is called an assignment edge.



→ Presence of a cycle in a resource-allocation graph is a necessary condition for a deadlock.

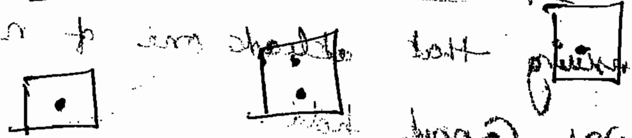
→



necessary and sufficient

$P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_1$ → $P_1$

→



deadlock.

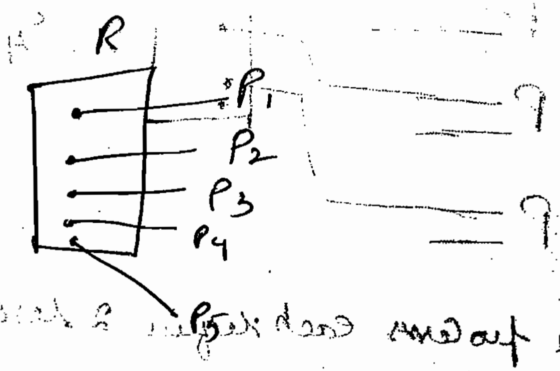
→ In summary, if a resource-allocation graph does not have a cycle, then ~~state~~ the ~~system may & may not be in a deadlock state.~~ system is not in a deadlock state.

On the other hand, if there is a cycle, then the system may & may not be in a deadlock state.

**Deadlock handling strategies:-**

1. Ignorance: we ignore the problem altogether, and pretend that deadlocks never occurs in the system.
2. Deadlock prevention:- It is a set of methods for ensuring that ~~atleast~~ one of necessary conditions cannot hold.
3. Deadlock avoidance It requires that the operating system be given in advance additional information concerning

→ Consider a system with 'n' processes and a single resource type 'R' with 5 instances. Each process '$P_i$' requires 2 copies of resource R To complete its execution. What is the maximum value of 'n' that can exist in system with a deadlock free situation.



I process = 2

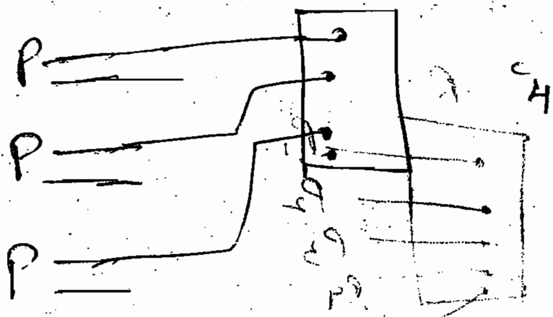
(n-1) process = 1

$$2 + (n-1) = 5$$

$$n + 1 = 5$$

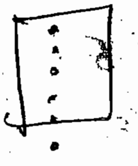
$$\Rightarrow n = 4$$

2) Conside a system with 3 processes and a single resource type 'R', each process 'Pi' requires 2 copies of 'R'. what is the minimum no of instances of 'R' in ~~due to~~ system with a deadlock free operation.



'4'.

3) n process each require 2 resources R = 6.
max no of processes so that no deadlock.



5

4) $n = 4$ and a resource ~~to~~ type R with many instances.

$P_1 = 10$ 9

$P_2 = 20$ 19

$P_3 = 8$ 7

$P_4 = 19$ 18

53 → max no of copies cause deadlock.

what is the minimum number of ~~to~~ instances of 'R' for a deadlock free operation.

5) A system has n-processes and a resource 'R' with each R 10-instances. Let each process requires '1' instance of resource R. what is the max number of processes that can be allowed to for a deadlock free operation.

n 200

no deadlock at all

## ~~Bauen~~ Deadlock Handlings

## Deadlock Prevention:

→ For a deadlock to occur, the four necessary conditions (ME, Hand W, No-preemption, circular wait) must hold. By ensuring that atleast one of these conditions cannot hold, we can prevent the occurance of deadlock.

### Mutual exclusion:

We cannot prevent deadlocks by denying the mutual-exclusion condition: some resources are intrinsically non sharable.

### Hold and wait:

Two protocols can be used:

a) one protocol, i.e. to ask each process to request and be allocated all its resources before it ~~begi~~ begins execution.

~~b) other protocols is to request ask a process to request resources only when the process has~~

b) other protocol is to allow a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

These two protocols have two main disadvantages.

a) Resource utilization may be low, since many of the resources may be allocated but unused for a long time.

b) Starvation is possible. A process that needs several popular resources may have to wait indefinitely, because atleast one of the resources that it needs is always allocated to some other process.

No Preemption :-

To ensure that this condition does not hold, we can use the following protocol.

If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all the resources currently being held are preempted.

Circular wait :-

Let $R = \{R_1, R_2, \ldots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows

# Deadlock avoidance

resource allocation state of system.
- available
- need of each process for every resource.

resource allocation state of system
- ~~available~~ no of processes ... Pn
- no of resources R1 --- Rm
  - SI
  - MI
- max demand made by each process for resource type
- Allocation made by OS to each process

**Ex 1** no of process = 3.

no of resources = 1 with 12 tape drives

|  | max | Allocation | Available | Need |
|---|---|---|---|---|
| P0 | 10 | 5 | 3 | 5 |
| P1 | 4 | 2 |  | 2 |
| P2 | 9 | 2 |  | 7 |

$< P_1, P_0, P_2 >$ → safe sequence

so safe state

Safe state :- A safe ~~it~~ state is safe, if the system can allocate resources to each process in some order and still avoid a deadlock.

Safe sequence :- A sequence of processes <P₁, P₂, ... Pn> is a safe sequence for the current allocation state, if for each $P_i$, the resources that $P_i$ can still request can be satisfied by the currently available resources plus the resources held by all [i<n] ~~all~~ $P_j$ with $j<i$.

Note :- A system is in a safe state, only if there exists a safe sequence.

→ Consider the following state :-

no of process = 3,

no of resources = 1 with 12 instances

| | max | Allocation | Available | Need |
|---|---|---|---|---|
| $P_0$ | 10 | 5 | 2 | 5 |
| $P_1$ | 4 | 2 | | 2 |
| $P_2$ | 9 | 3 | | 7 |

no safe sequence, so it is not in safe state.

-> The ~~algorithm~~ general algorithm used for deadlock avoidance is known as ~~bankers~~ banker's algorithm

-> Data structures used of bankers algorithm are;

**Available:** A vector of length 'm' indicates the number of available resources of each type. If available [j] = K, there K instances of ~~resource~~ type $R_j$ available.

**Max:** An n x m matrix defined the maximum demand of each process. If Max [i, j] = K, then process $P_i$ may request at most K instances of resource type $R_j$.

**Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation [i, j] = K, then process $P_i$ is currently allocated 'K' instances of resource type $R_j$.

**Need:** An $n \times m$ matrix indicates the remaining resource need for each process. If Need$[i, j] = K$, then $P_i$ may need 'K' more instances of resource type $R_j$ to complete its task.

**Note:** Need$[i, j]$ = max$[i, j]$ − Allocation$[i, j]$

**Problem:** Consider a system with 5 processes $P_0$ through $P_4$ and three resource types A, B, C. Resource type A has 10 instances and resource type B has 5 instances and resource type C has 7 instances. Suppose that, at time $T_0$, the following snapshot of the system has been taken:

|  | Allocation<br>A B C | Max<br>A B C | Available<br>A B C |
|---|---|---|---|
| P0 | 0 1 0 | 7 5 3 | 3 3 2 |
| P1 | 2 0 0 | 3 2 2 |  |
| P2 | 3 0 2 | 9 0 2 |  |
| P3 | 2 1 1 | 2 2 2 |  |
| P4 | 0 0 2 | 4 3 3 |  |

So derive the need as follows:

**Need**

7 4 3

P1 2 2

6 0 0

P3 0 1 1

P4 4 3 1

Safe ~~set~~ sequence <P1, P3, P4, P2, P0>

Now: P1 request (1,0,2). Can this request be granted? will this lead to an unsafe state?

To check this, let us assume that we have granted the request, then the system will will change to another state.

|  | Allocation |  |  | ~~Max~~ Need |  |  | Available |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 2 | 7 | 4 | 3 | 2 | 3 | 0 |
| P1 | 3 | 0 | 2 | 0 | 2 | 0 |  |  |  |
| P2 | 3 | 0 | 2 | 6 | 0 | 0 |  |  |  |
| P3 | 2 | 1 | 1 | 0 | 1 | 1 |  |  |  |
| P4 | 0 | 0 | 2 | 4 | 3 | 1 |  |  |  |

Now check if ~~the~~ ~~st~~ this new state is safe & we found a safe sequence <P1, P3, P4, P0, P2> and so this state is safe.

Now a new request for P0 for (0,2,0) can be granted? check if this can be granted / not granted

It cannot be granted because there is no safe sequence.

Ex: Allocation / requirement / available

|  | A | B | C | D | A | B | C | D |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | 1 | 2 | 2 | 1 | 3 | 3 | 2 | 2 |
| $P_2$ | [illegible] | [illegible] | 3 | 3 | 1 | 2 | 3 | 4 |
| $P_3$ | 1 | 1 | 1 | 0 | 1 | 1 | 5 | 0 |

available

| A | B | C | D |
|---|---|---|---|
| 3 | 1 | 1 | 2 |

Need

|  | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 1 | 0 | 1 |
| $P_2$ | 0 | 2 | 0 | 1 |
| $P_3$ |  |  | 4 | 0 |

<$P_1$, $P_2$, $P_3$> is safe sequence.

so this is safe state.

Now $P_2$ request for (0,1,0,0). It leads to unsafe state

Now $P_3$ reques for (0,0,2,0). It cannot be met.

> $P_3$ asks for (0,0,1,0). Safe

($P_1$, $P_2$, $P_3$).

→

| | Allocation<br>A B C D | max<br>A B C D | Available<br>A B C D |
|---|---|---|---|
| $P_0$ | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| $P_1$ | 1 0 0 0 | 1 7 5 0 | |
| $P_2$ | 1 3 5 4 | 2 3 5 6 | |
| $P_3$ | 0 6 3 2 | 0 6 5 2 | |
| $P_4$ | 0 0 1 4 | 0 6 5 6 | |

1 5 2 0
0 0 1 2 → Po
1 5 3 2
1 3 5 4
2 8 8 6

Need

| | A | B | C | D |
|---|---|---|---|---|
| Po | 0 | 0 | 0 | 0 |
| $P_1$ | 0 | 7 | 5 | 0 |
| $P_2$ | 1 | 0 | 0 | 2 |
| $P_3$ | 0 | 0 | 2 | 0 |
| $P_4$ | 0 | 6 | 4 | 2 |

If a request from process $P_1$ arrives for (0,4,3,0

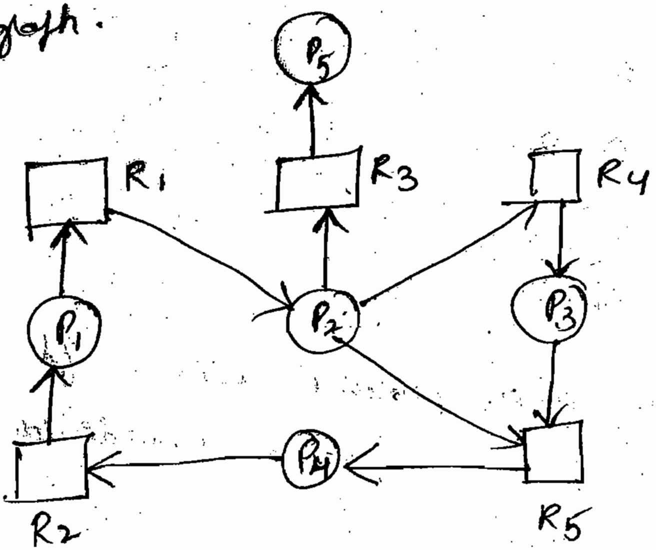
Can the request be granted immediately

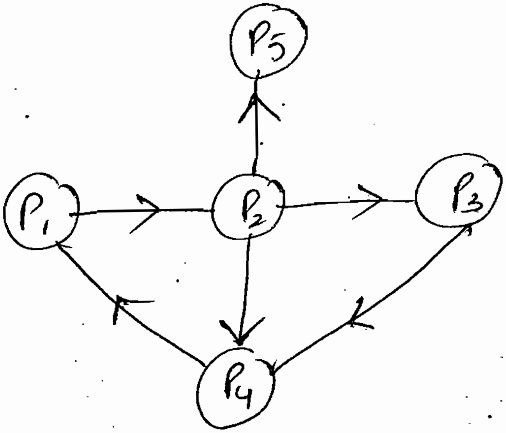
## Deadlock detection and recovery:

- If a system does not employ a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.
- In such a situation, we need to detect the deadlock and recover from it.

Case (i): when all resources are of single instance type

→ In this case, we use a variant of the resource allocation graph called a wait-for graph.





~~As~~ There will be a deadlock in the system if and only if the wait-for graph contains a cycle.

Case 2: Several instances of a resource type:

when each resource has several instances, then we have to check for all / possible if there is safe sequence or not.

## Recovery from Deadlock:

### a) Process termination:

In this method, we eliminate deadlocks by aborting a process and reclaiming all resources allocated to the terminated process. we can follow two strategies here:

(i) Abort all deadlocked processes:

advan : Simple to implement

disadva : whatever process have completed till now, will be lost.

(ii) Abort one process at a time until the deadlock cycle is eliminated.

advan : all the processes need not be aborted. so some computations will be saved.

disad : Implementation is not simple, we have to abort each process and check if the deadlock is eliminated or not.

## Resource Preemption:

To eliminate deadlocks, using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then.

In this method, three issues need to be addressed

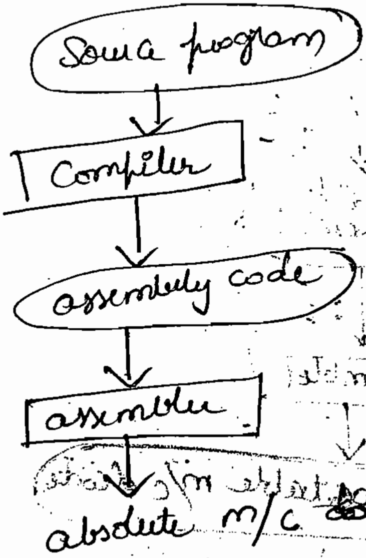
1) Selecting a victim:
Which resource and which process?

2) Roll back:
what should be done with that process?

3) Starvation:
How do we ensure that starvation will ~~not~~ occur?

# memory management:

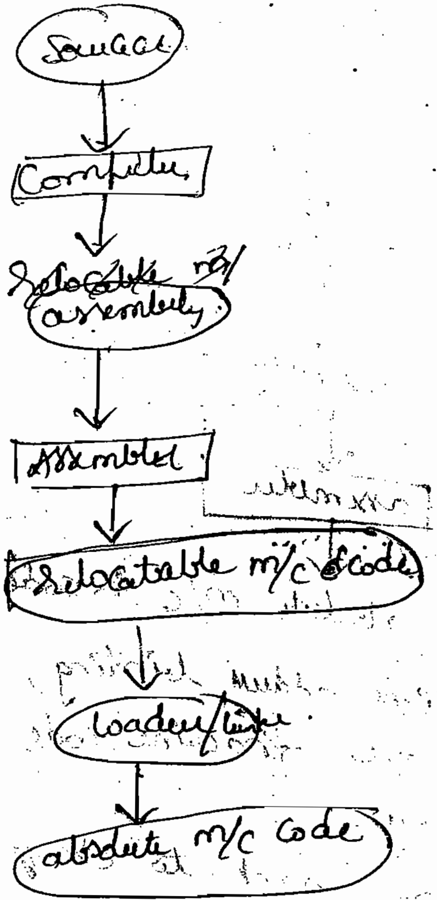
## compile time address binding:



For compile time address binding, we should know where the machine code will be loaded in the memory.

? Disadv: If we want to change the starting memory location, we will have recompile the source code.

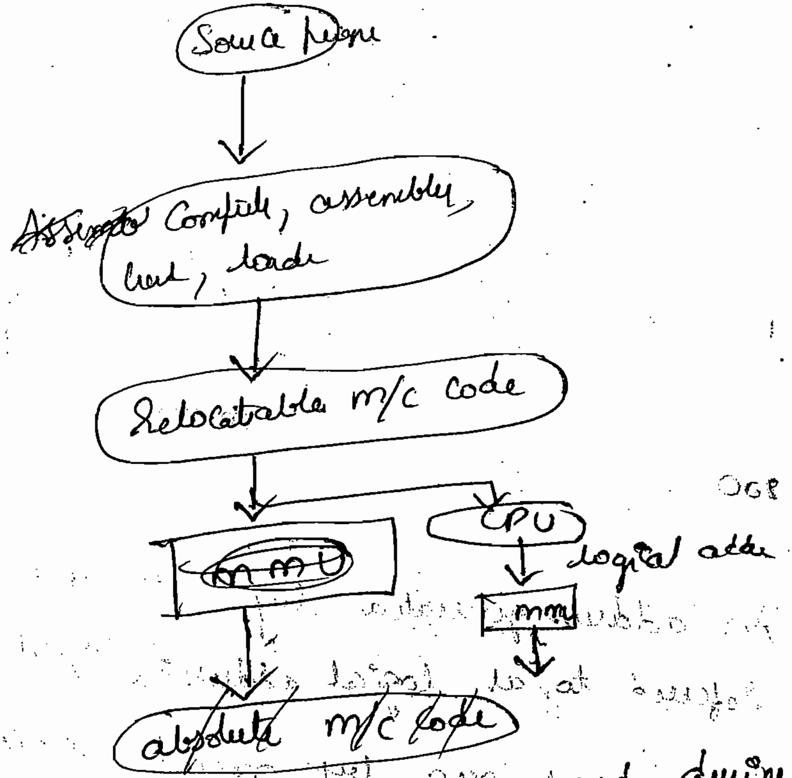
# Load time address binding:



~~If After~~ Compile time

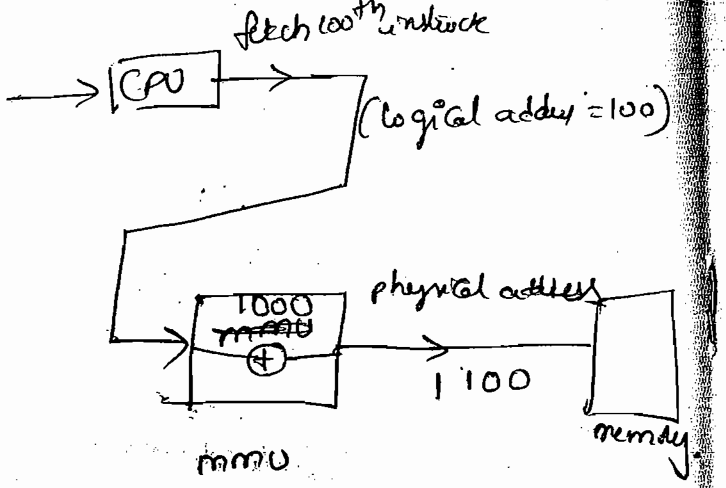
In load time binding, if the starting address changes, we need only to reload the exe code to incorporate this changed value.

Execution time address binding or runtime address binding :



→ If the process has to be moved, during its execution from one memory segment to another, then binding must be delayed until run time.

```
1
2
10 CMP R0, 1
11 JE  100
12
.
.
100 GOTO 200
.
.
200
```



→ An address generated by the CPU is commonly referred to as logical address of logical address.

→ An address seen by memory unit & referred to as physical address.

→ In case of compile time and load time binding

logical address = physical address.

→ The runtime mapping from virtual to physical addresses is done by a h/w device called the memory-management unit (MMU).

Dynamic loading:

→ A routine is loaded into memory only when it is called.

→ All routines are kept on disk in a ~~Seleo~~ relocatable load format.

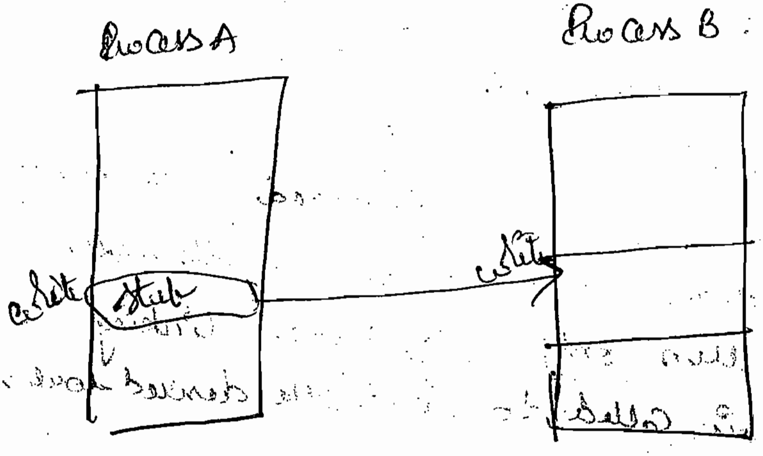
→ The main program is ~~ø~~ loaded into memory and executed. when a routine calls another routine, then only relocatable linking loaded is called to load the desired routine

Advantage:-

An unused routine is never loaded.

Note: Dynamic loading does not require special support ~~fo~~ from the operating system. It is the responsibility of users to design their programs to take advantage of such a method. Operating system may ~~pro~~ help the programmer, however, by providing library routines to implement dynamic loading.

# Dynamic linking and shared libraries :-



Here linking is postponed until run time.

→ A stub is included in the m/c code for each library routine reference.

→ This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine, & how to load the library if the routine is not already present.

→ Stub replaces itself ~~by the~~ with the address of address of routine and executes the routine.

→ under this scheme, all processes that use a language library execute only one copy of the library code.

→ unlike dynamic loading, dynamic linking generally requires help from the OS.

→ If the processes in memory are protected from one another, then OS is the only entity that can check to see whether the needed routine is in another processes memory space & that can allow multiple processes to access the same memory address.

→ <u>overlays</u> :-

→ we use overlays to store a process that is larger than the amount of memory allocated to it.

→ The idea of overlays is to keep in

memory only those instructions and data that are needed at any given time.

→ when other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

Ex: Consider a two pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code, using symbol table. We can partition such an assembler into pass 1 code, pass 2 code, the symbol table and common support routines used by both pass 1 and pass2. Let us assume that sizes of these components are as follows:

| | |
|---|---|
| Pass1 | 70 KB |
| Pass 2 | 80 KB |
| Symbol table | 20 KB |

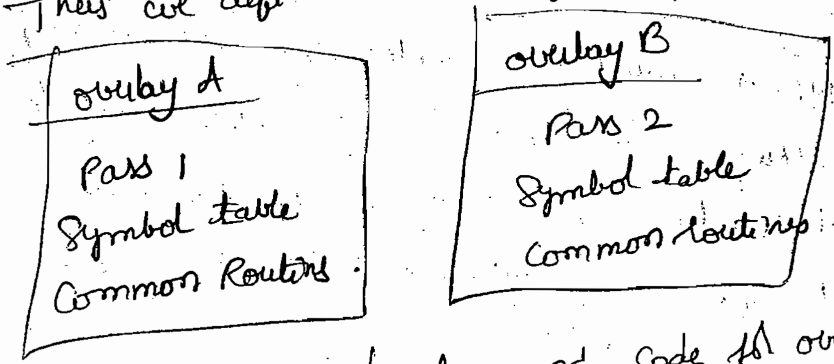
Common routines 30 KB.

To load everything at once, we would require 200 KB of memory. If only 150 KB of memory is available, we cannot run our process.

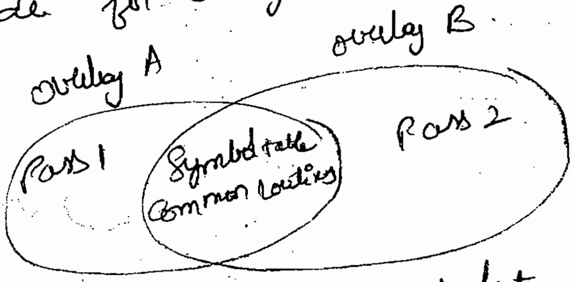
→ Hence we use an overlay driver to load pass 1 code, then pass 2 code.

→ The codes for

→ Then we define two overlays:



→ code for overlay A and code for overlay B



are kept on disk as absolute memory images and are read by the overlay driver as needed.

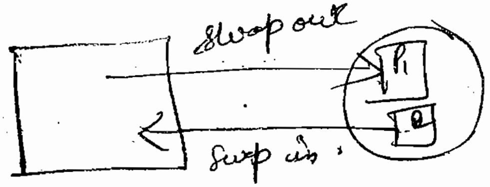
→ As in. dynamic loading, overlays do not require any special support from the OS.

→ They can be implemented completely by user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The OS ~~only~~ notices only that there is more I/O than usual.

→ overlays is currently limited to microcomputer and other systems that have limited amounts of physical memory and that lack ~~to~~ h/w support for more advanced techniques.

## Swapping:

Swapping is moving a process b/w main memory and secondary memory.



→ If we use Compile time & load time binding, then process must be ~~loada~~ swapped into the same location, i.e., the process ~~If we use~~ cannot be moved to different locations.

→ If we use execution time binding, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

→ Dispatcher is responsible for swapping in and swapping out.

→ Context switching time in a system which ~~int~~ involves swapping is fairly high.

Ex: Let size of a process be 1MB.
and the disk has a transfer rate of 5MBPS.
Transfer of 1MB ~~fr~~ to & from memory takes
$= 1/5$ seconds
$= 0.2$ seconds
$= 200$ milli seconds.

So both swap in and swap out takes 400 milliseconds.

So for efficient CPU utilization, we want the execution time for each process to be long relative to the swap time.

memory manager

functions
1. Allocation
2. Protection
3. Free space management.
4. De allocation.

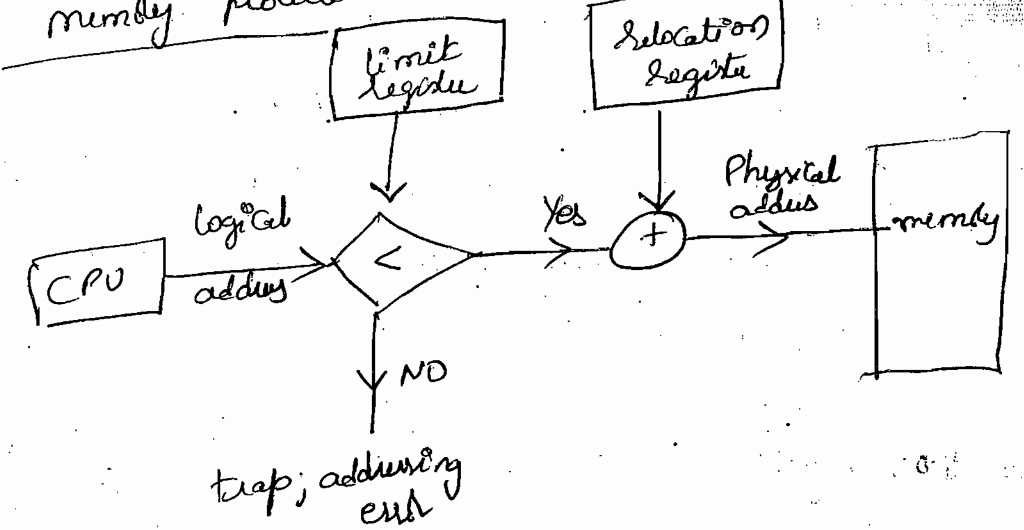
Goals
1. Efficiency.
2. ability to run large programs in small amounts of memory.

Techniques

Contiguous
- overlays
- Partitioning
  - fixed
  - variable
- Buddy

Non-contiguous
- Paging
- Segmentation
- Virtual memory.

→ memory protection :-



Each logical add

→ Protecting the OS from user process and user processes from one another is needed

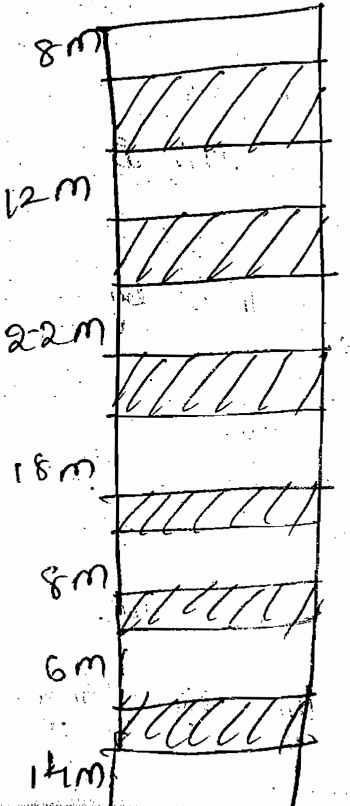
→ we can provide this protection by limit and relocation registers.

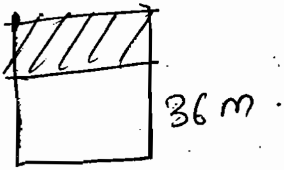
→ Each logical address should be less than the limit register

→ The MMU converts the logical address to physical address dynamically by adding the value in the Relocation Register.

→ When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with correct values as part of the context switch.

## Partitioning :





1) $P_i$ → 14

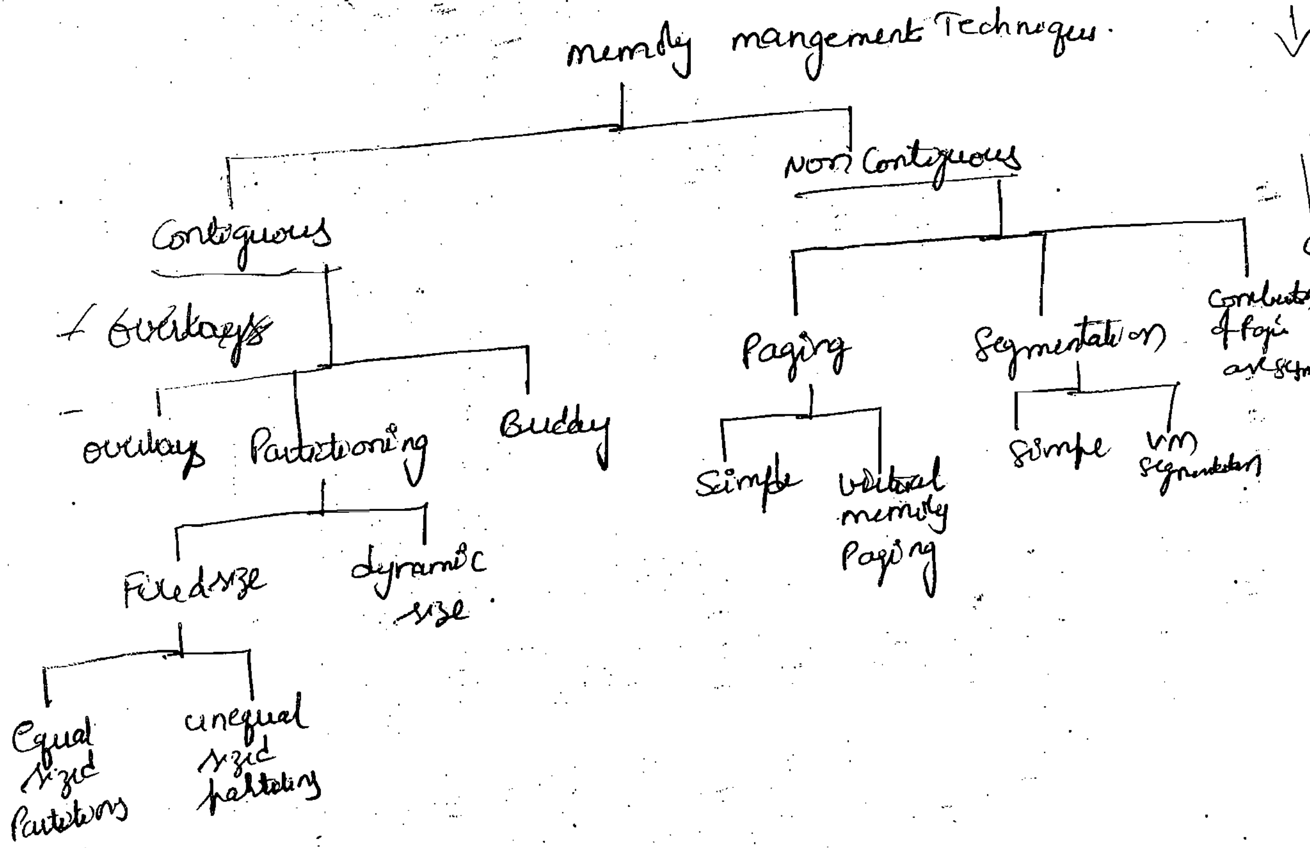
(i) First - 22m

(ii) Best - 14m

(iii) Next -

(iv) worst - 36 m.

2) How many successive requests of size 5m could be satisfied 22.



memory allocation

Partitioning :

→ user space can be partitioned ~~into two~~ into fixed size partitions and variable size partitions.

Fixed size partitioning : (MFT)

→ In this scheme, memory (user space) is partitioned into partitions, ~~whose size~~ and is fixed/ the size of each partition does not change.

→ In order to run a process, it has to be placed in a partition which can hold it.

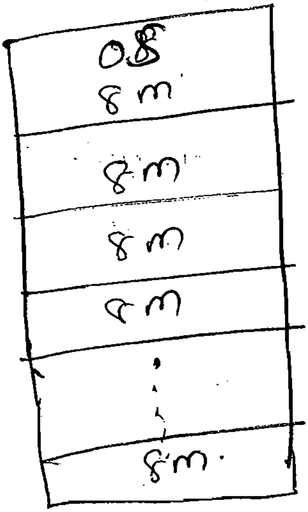
→ ~~If the sizes of all partitions are equal~~, ~~and~~ (decided at system generation time)

→ Since the size of each partition is fixed X and the size of memory is fixed, we can have only fixed number of processes at any time in the memory.

→ Therefore, degree of multiprogramming is fixed in this scheme.

→ ~~St~~ So, it is also called multiprogramming with fixed number of tasks (MFT).

→ If the sizes of all partitions are equate, then it is called, equal sized <u>fixed</u> partitioning



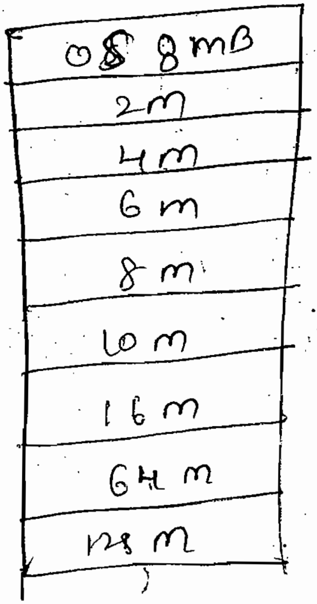
→ ~~If If the prog process is too big~~ to ~~fit into a~~ partition,

→ Any process can be placed in any partition, since, all partitions are of equal size

→ If a process is too large to fit in a partition then overlays should be used (overlays is burden on user side).

→ If the process is smaller than the size of a partition, then is some space wasted due to inside the partition, which cannot be used by any other process. This is called internal fragmentation.

→ Both overlaying and internal fragmentation, can be decreased, though not solved, by using unequal size partitions.



→ In unequal size partitioning, ~~the~~ ~~more~~ overlays will be used only if ~~the~~ ~~biggest~~ largest partition also can't hold a process.

→ Internal fragmentation can be decreased by choosing a ~~partition~~ smallest empty partition, which can hold a process. (Best fit) This placement algorithm is called best fit algorithm.

→ In some situations, (if the process is very large and) if we add all the empty ~~available~~ partitions sizes, and the memory wasted ~~available~~ by ~~external~~ internal fragmentation, then ~~also~~ we can accomodate the process. But since this memory is scattered all around, we won't be able to accomodate ~~all~~ ~~the~~ the process in memory. This phenomenon is called external fragmentation.

Dis advantages of fixed partitions:-

a) Internal fragmentation
b) External fragmentation
c) degree of multiprogramming is ~~fix~~ fixed and is limited by the number of partitions in the memory.

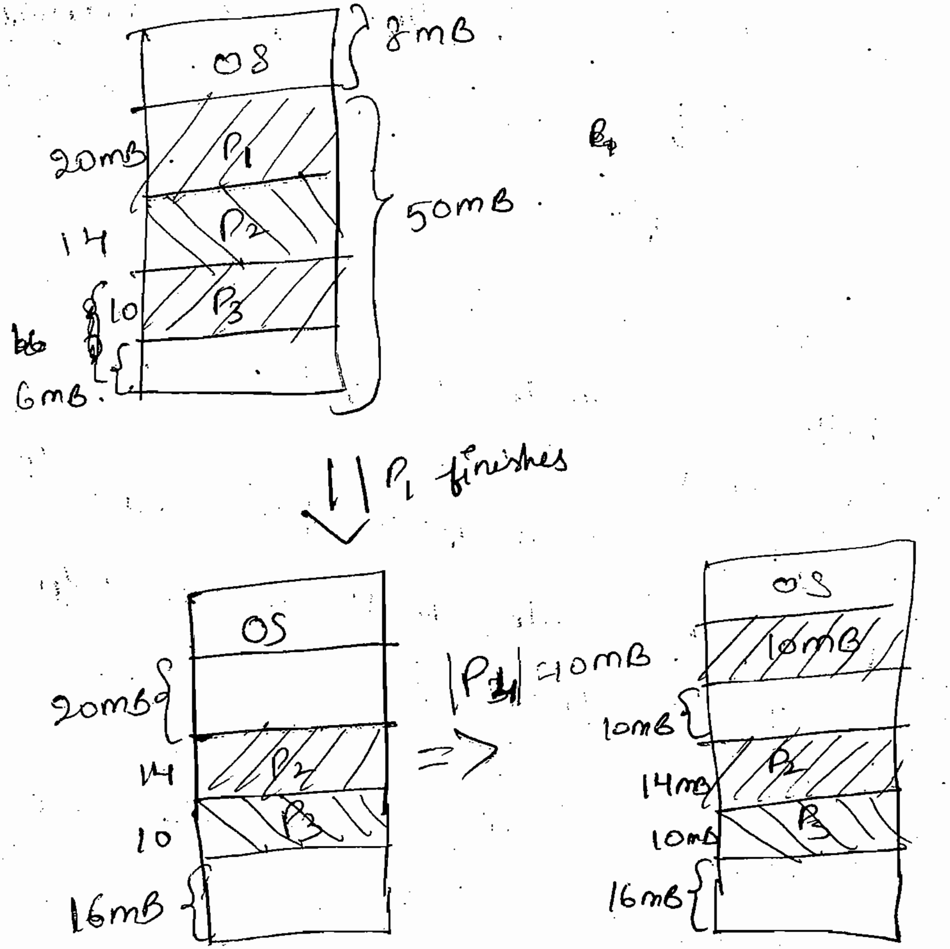
→ To avoid the disadvantages (a) and (c) ~~of~~ the fixed partitioning, we go for dynamic & variable sized partitioning.

Dynamic & variable sized partitioning:- (MVT)

→ In this partitioning, the no of partitions and size of each partition is not fixed at the system generation time.

→ Partitions are created dynamically, with the size of each ~~process~~ partition equal to the size of the process loaded. ~~A~~ Therefore, degree of multiprogramming is variable. Hence it is also called multiprogramming with variable number of tasks (MVT).

Ex:



→ In this scheme, at any time, there will be a set of holes, of various sizes scattered throughout memory.

→ whenever a request arises to load a process, which hole has to selected is main issue here.

2. The allocation can follow four stategies,

a) First fit  b) Best fit  c) worst fit

d) Next fit.

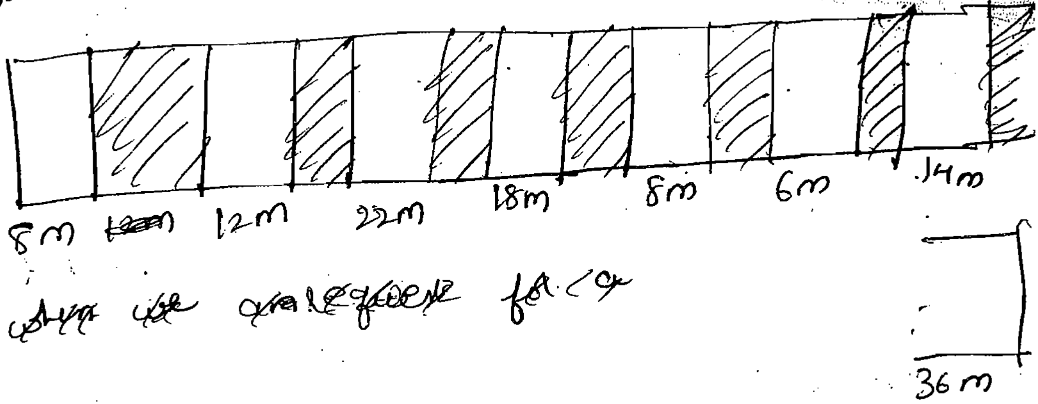
First fit: ~~Sear~~ search from starting of the list of holes, and allocate the first hole that is big enough.

Next fit: Same as first fit, but instead of searching from ~~first~~ starting of the list of holes, search the list from where you previous search ended.

Best fit: Allocate the smallest hole that is big enough to hold the process.

worst fit: Allocate the largest hole

Consider a MVT system with following properties of memory



→ [illegible]

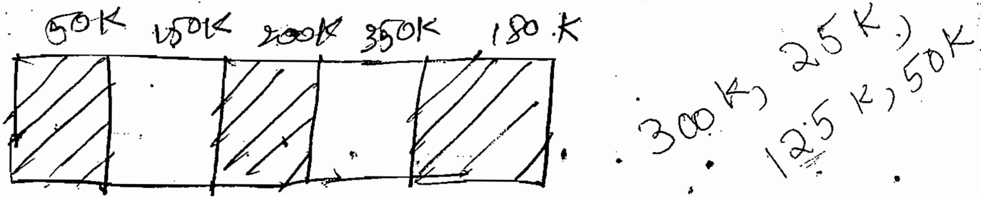
→ when a process request for 14MB, where will it be placed using

(i) First fit — 22m
(ii) Best fit 14m.
(iii) Next fit
(iv) worst fit — 36

→ How many successive requests of size 5M could be satisfied.

Ans: 21

→ Consider the MVT with following snapshots



The request could be satisfied with which of the following:

a) Both first and best  b) neither first nor best.

✓c) First fit but not best fit

d) Best fit but not first fit.

Advantage of MVT :-

a) degree of multiprogramming is not fixed.

b) No internal fragmentation.

Disadvantage of MVT is external fragmentation.

external / fragmentation

→ one solution to the problem of external fragmentation is compaction.

→ Compaction is to shuffle the memory contents to place all free memory together in one large block.

→ Compaction is possible only with dynamic address binding.

→ Another possible solution to the external fragmentation problem is to permit the process to be non contiguous (Paging, segmentation and combination).

**Paging :**

Basics)

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1024 = K$

$2^{20} = M$, $2^{30} = G$, $2^{40} = T$

$2^{50}$ – Peta

$2^{60}$ – Exa

$2^{70}$ – Zetta

$2^{80}$ – yotta

$2^{32} = 2^{30} \times 2^{2} = 4 \times 2^{30} = 4 GB$.

$2^{45} = 2^{5} \times 2^{40} = 32 \times 2^{40} = 32 TB$.

$2^{16} = 2^{6} \times 2^{10} = 64 KB$

→ ~~If you want to address two words then no of bits needed.~~

→ word: Smallest addressable unit in memory is called a word.

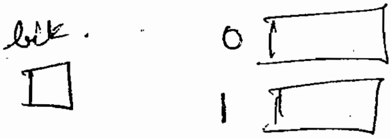
~~A word can be one byte, two bytes &~~

A word ~~Can be~~ is a collection of one & more bytes.

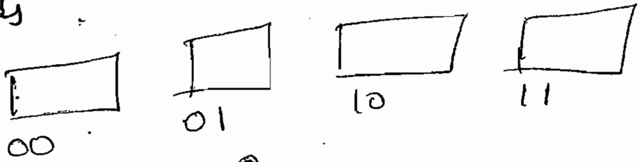
→ ~~The collection depends~~

→ The size of a word generally depends on the architecture of the computer.

→ How many bits are need to address 2words.



→ How many bits are needed to address 4 words .



→ 8 words - 3 bit addr

16 words - 4 bit add .

32 " - 5 bit address .

~~n words~~ - ~~log~~

33 words - 5 ✗

6 address .

6 address - $6 \; 2^6 - 64$

$$\begin{array}{r} 64 \\ 33 \\ \hline 31 \end{array} \text{ - wasted}$$

→ 64 K words - $2^6 \times 2^{10} = 2^{16}$. K M G T P

10 20 30 40 50.

16 bits are required

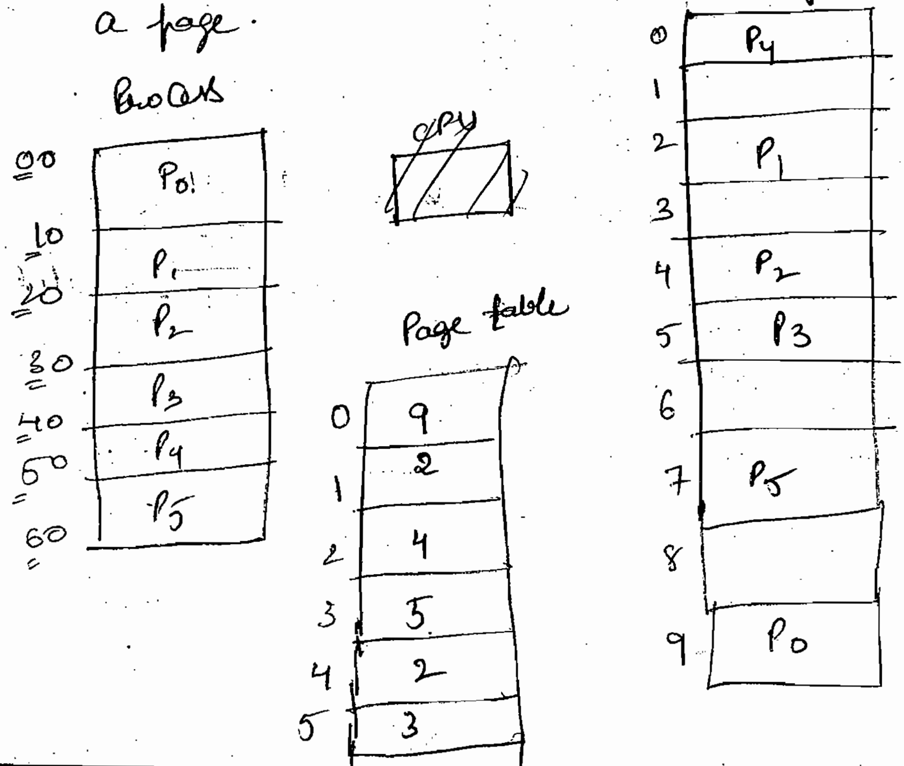
128 MW = $2^7 \times 2^{20} = 2^{27}$ 27 bits .

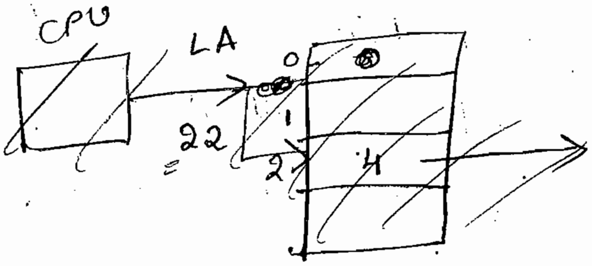
## Paging:

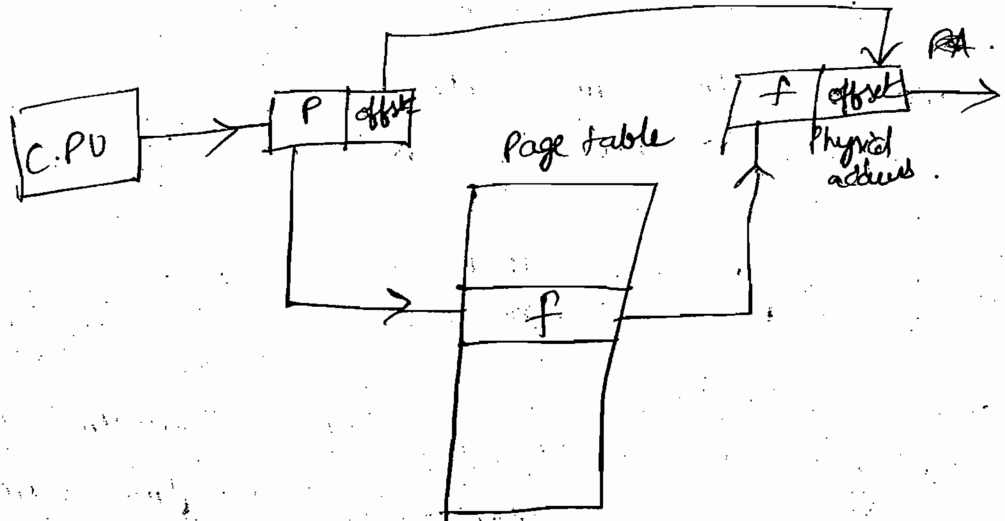
→ Paging is a memory management scheme that permits the physical address space of a process to be noncontiguous.

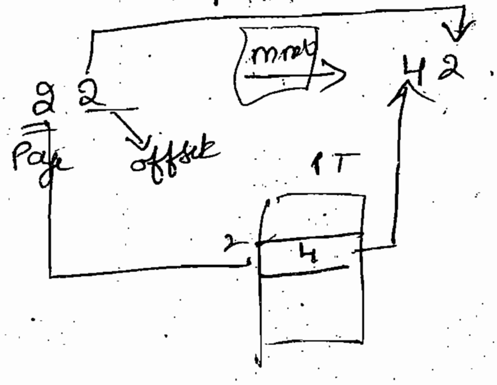
→ In this scheme, process (logical address space) is divided in pages and memory is divided into frames.

→ Size of a frame is equal to the size of a page.




CPU
LA
0
1
22
2
4


C.PU
P
offset
Page table
f
f
offset
PA
Physical address


22
Page
offset
mref
42
PT
2
4

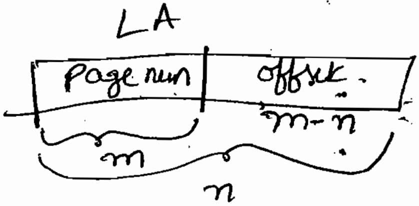
> advantages :-

No external fragmentation, no need of

> Before executing a process, all its pages will be loaded into any available memory frames.

> Logical address generated by CPU, will be divided into two parts

a) Page number b) offset.



-> using page number, page table is consulted and frame number is obtained. Now, frame number and offset forms PA.

-> Let size of a process be $2^n$ ~~bytes~~ words.

size of LA = n bits

size of ~~one~~ a page be $2^k$ words.

offset contains = k bits.

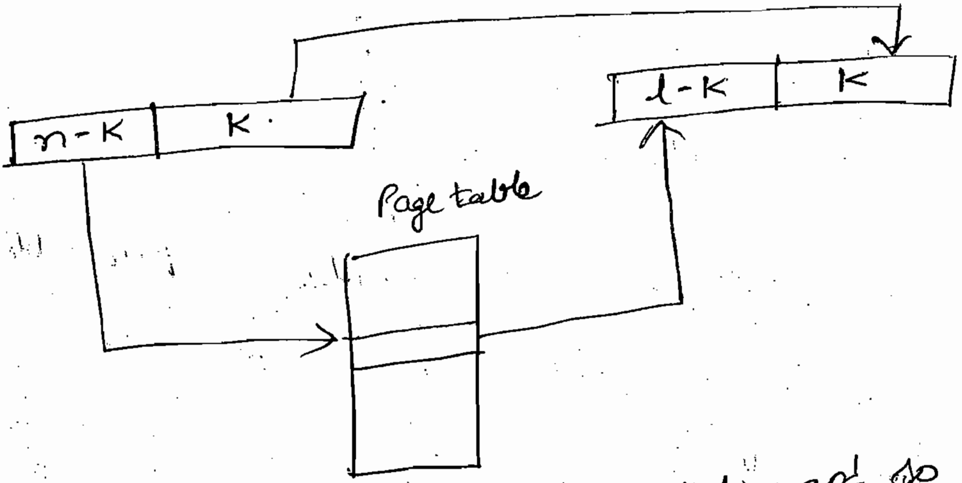
no of pages in process = $\frac{2^n}{2^k} = 2^{n-k}$.

Pagenumber field contains n-k bits

→ Let size of main memory be $2^l$ words.

$PA = l$ bits.

no of frames in memory $\frac{2^l}{2^k} = 2^{l-k}$.



→ There is no external fragmentation and so no need of compaction.

→ There may be internal fragmentation, if the size of a process is not a power of 2.

→ Average internal fragmentation = $\left(\frac{\text{Page size}}{2}\right)p$.

→ Proof) average wastage per process = $\frac{\text{page size}}{2}$

→ so smaller the page size, lesser the internal fragmentation.

→ Page table size of a process $\propto$ no of pages of the process.

→ ~~larger the no of pages/process~~ ,

→ no of pages of process $\propto \frac{1}{\text{Page size}}$.

$\therefore$ Page table size $\propto \frac{1}{\text{Page size}}$.

→ larger the page size, smaller the page table size.

→ so page size should not be too large nor too small.

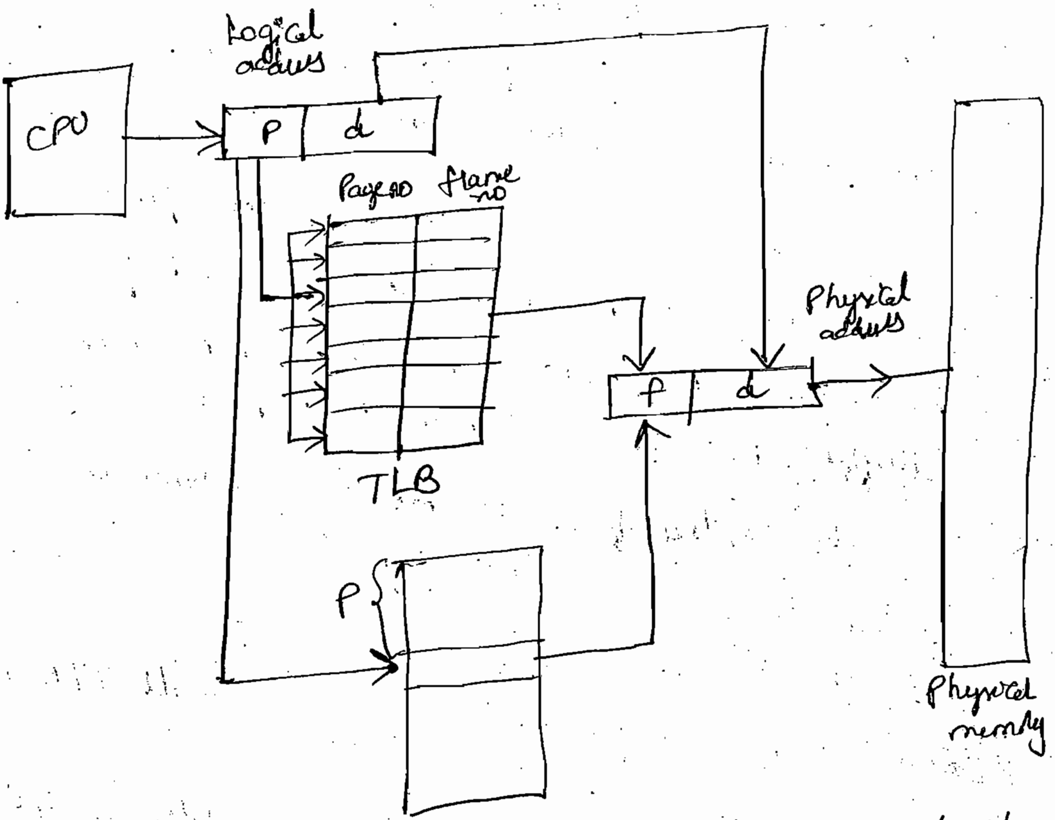
→ The operating system will maintain a frame table. The frame table has one entry for frame. This

→ ~~This~~ gives the information like which frame is allocated to which process & if the frame is free etc.

→ Paging increases context-switch time, because the page table has to be loaded into memory.

Hardware support for paging:

- -> Pagetable can be placed in a set of registers.
- This is preferable if the pagetable size is ~~too small~~ reasonably smaller.
- -> The ~~best~~ standard solution to this problem is to use a special small, fast lookup hardware cache, called translation look-aside buffer (TLB). (associative, high-speed memory).
- -> Each entry of a ~~TLB~~ TLB contains a ~~tag~~ tag and a value.
- -> ~~But the size of~~
- -> But TLB is expensive and so very small TLB is used in many systems.
- -> So only a part of ~~TLB~~ ~~is~~ page table is kept in TLB. Each entry of TLB contains some page number as tag and frame number as value.
- -> When the LA is generated by CPU, its page number is presented to the TLB.



→ If page number is ~~not~~ ~~in~~ TLB found, its frame number is immediately available and is used to access memory.

→ If Pagenumber is not in TLB (known as a TLB miss), ~~a memory reference to~~ the page table ~~must be made~~ in the memory will be consulted.

→ The page table entry which caused the miss will be again added to TLB, so that they will be found quickly on the next reference.

Hit ratio :-

→ The percentage of times that a particular page number is found in the TLB is called the hit ratio.

→ If hit ratio = 70, it means out of hundred time, we found a page 70 times

→ Problem :-

Consider a system ~~with~~ in which it takes 20 nanoseconds to search TLB, and 100 nano seconds to access memory. TLB has a hit rate of 80%. what is the effective memory access time.

Sol: Here we have two cases,

(i) TLB HIT

(ii) TLB miss.

TLB hit : 20 ns to search TLB

+

100ns to fetch desired word.

= 120 ns.

TLB miss : 20 ns to search TLB

+

100ns to access page table in memory for frame number

+

100 ns to ~~f~~ fetch the desired word.

= 220 ns.

To find effective memory access time, we must weigh each case by its probability.

effective access time = 0.80 × 120 + 0.20 × 220

= 140 nano seconds.

if hit ratio is 90 % 0.90 × 120 + 0.10 × 220

=

98%

$0.98 \times 120 + 0.02 \times 220$

Problems:-

→ If LAS = 8KB, PAS = 4KB and page size = 1KB.

(i) size of logical address and no of bits in each field.

(ii) size of physical address and no of bits in each field.

(iii) If Each entry size of each entry of page table is 'e' bytes, then the PT size = ~~(N×e bytes~~.

→ If ~~LA = (29~~ size of logical address is 29 bits, no of pages = 2K ~~and~~ no of frames is 256, ~~Calculate PA~~, and each addr contains 4 bytes, then calculate

a) LA and LAS in bytes  b) PA PAS in bytes

c) If each PTE is 4 bytes, then calculate PTS in bytes

LA = 29 bits.

So LAS = $2^{29}$ words

= $2^{29} \times 4$ bytes.

PAS = no of frames × frame size.

$\qquad\qquad\qquad\qquad\qquad\qquad ||$

$\qquad\qquad\qquad\qquad\qquad$ Page size.

LAS = no of pages × page size.

⇒ no of pages =

⇒ Page size = $\frac{\text{LAS}}{\text{no of pages}}$

$= \frac{2^{29}}{2K} = \frac{2^{29}}{2^{11}} = 2^{18}$ words

PAS = ~~no~~ $256 \times 2^{18}$ words.

$= 2^{8} \times 2^{18}$ words

$= 2^{26}$ words $= 2^{26} \times 4$ bytes

$= 2^{28}$

Page table size = no of entries in pagetable × size of each entry.

= no of pages × size of each entry.

= $2K \times 4$ Bytes

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

= 8 KB.

→ If LAS = 8GB, PAS = 64MB and PS = 16KB.

Calculate a) PA size. b) LA size c) PT size.

~~No of Pages = $\frac{64MB}{16KB}$ = 4K~~

2
33
14
19

~~No of frames = 64M~~

No of pages = $\frac{8GB}{16KB} = \frac{2^3 \times 2^{30}}{2^4 \times 2^{10}} = \frac{2^{33}}{2^{14}}$

$= 2^{19}$

26
14
12

No of frames = $\frac{64M}{16K} = \frac{2^6 \times 2^{20}}{2^4 \times 2^{10}}$

$= 2^{12}$

Frame number field = 12 bits.

PT entry size = 12 bits

PT size = no of ~~pages~~ entries × size of each entry

= no of pages × size of each entry

= $2^{19} \times 12$ = $12 \times 2^{19}$ bits

= $12 \times 2^{16}$ bytes.

> LA = 32 bit, PS = 4K, PAS = 64 mB

then PT is bytes.

<u>Restrictions</u>:

→ Some bits are stored in page table entry along with frame number, these the page information

→ Read-write or read-only bit indicates if the page can be used only for reading or it can be written also

→ A valid-invalid bit say if the address generated by CPU is valid or not generated

## Hierarchical paging:

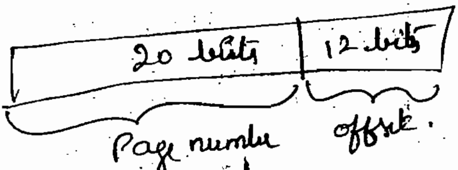
→ If the page table of a process is too large, we would not be able to allocate the page table contiguously in main memory.

→ In such case, the page table will be divided into ~~pa~~ blocks of equal size.

Ex: ~~Let LAS=2?~~

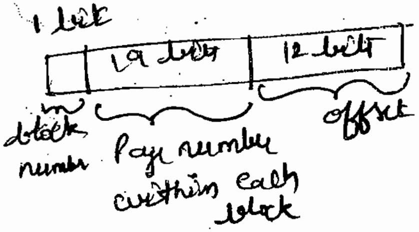
Consider a system with 32 bit logical address space. Page size is 4KB. Then

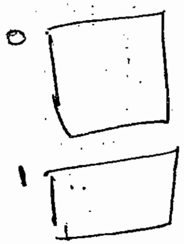
Page table / will have



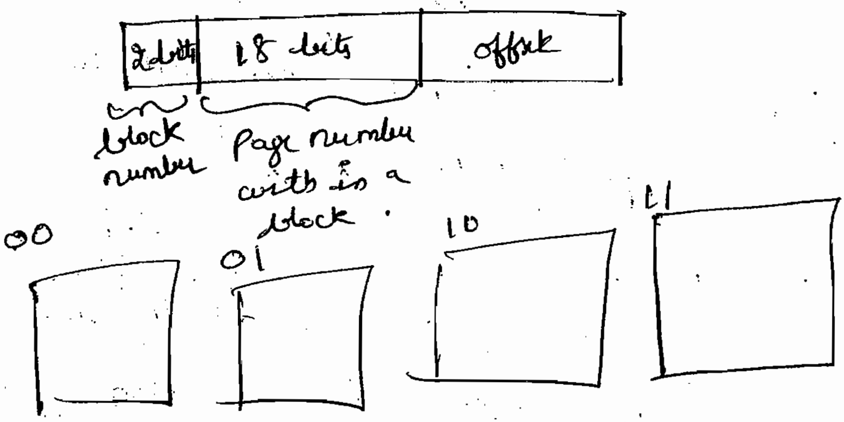
Page table will have $2^{20}$ entries.

If we divide it into two parts,





Divide the page table into 4 - blocks, then LA can be divided as



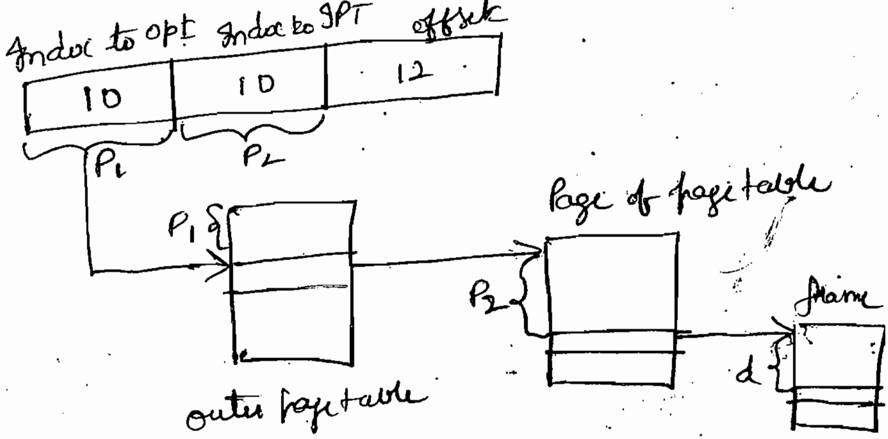
→ If each page table entry is 4 bytes, then

PT size = $2^{20} \times 2^{2} = 2^{22}$ bytes.

→ If we splits the page table in such a way that each block fits in a page, then we will get $\frac{2^{22}}{2^{12}} = 2^{10}$ pages in the page table.

→ Then we have a ~~page table called~~ two page tables called outer page table and inner page table.

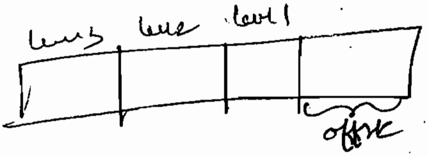
→ now LA looks like this



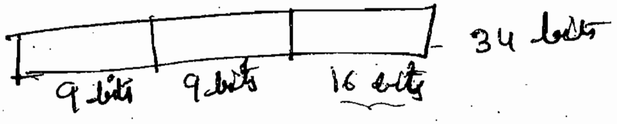
⇛ → This is called two level paging scheme.

→ we can go upto any levels of paging

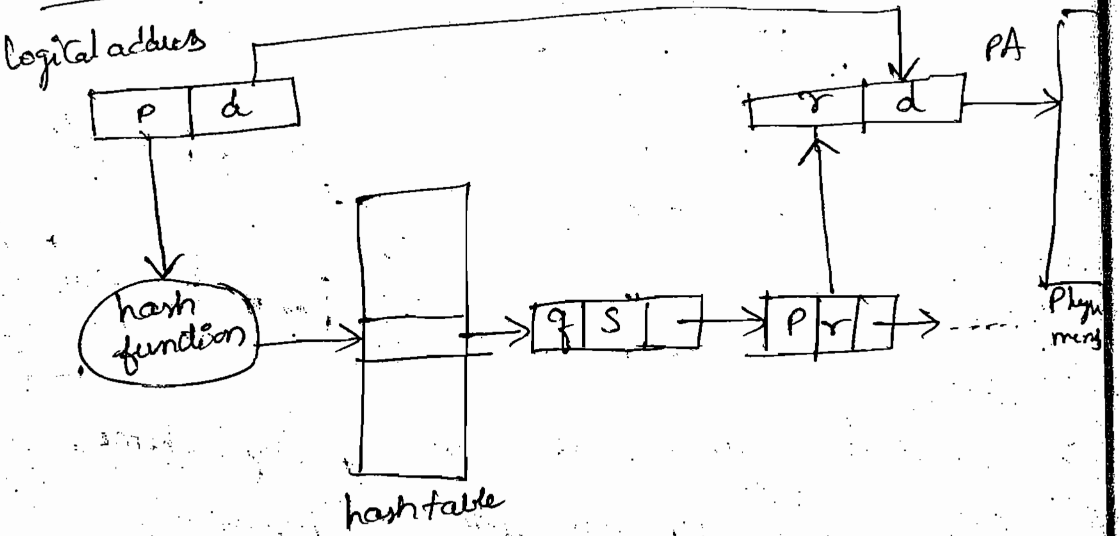
Ex :



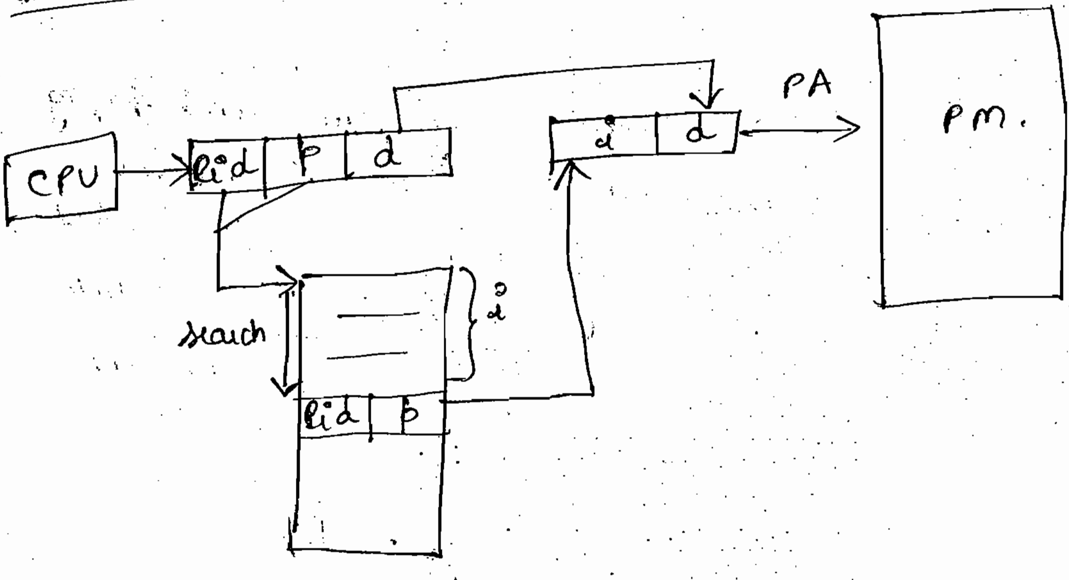
→ Consider a system using two level paging in which inner page table is divided into 512 pages each of size 512 words. If the length of logical address is 34 bits, Calculate the size of PAS if there are 256 frames in it.



## Hashed page tables:-



## Inverted page table:-



one drawback with page table is that, each process will have a page table and each page table will contain millions of entries, which occupies large amounts of PM.

→ To solve this problem, we use an inverted page table.

→ An inverted page table has one entry for each real page (or frame) of memory.

→ Each entry has consists of the virtual address of page store in that real memory location, with information about the process that owns that page.

→ Thus, only fa one page table is in the system, and it has only one entry for each page of PM.

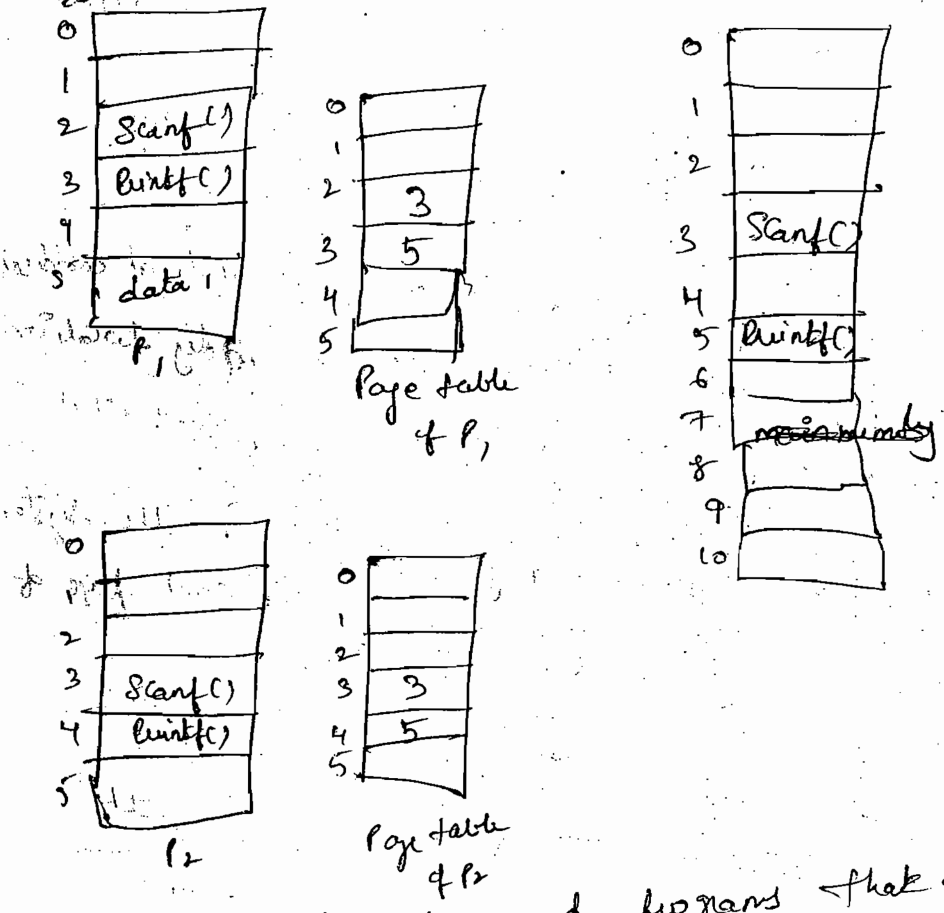
⇒ Shared pages:-

→ Another advantage of shar paging is the possibility of sharing common code.

⇒ Reentrant code (or pure code) is a non-self-modifying code.

→ If the code is reentrant, then it never changes during execution.

→ Thus, two or more processes can execute the same code at same time.

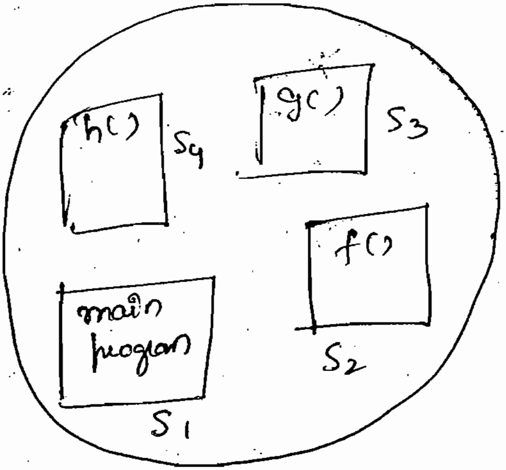


→ Examples of heavily used programs that are commonly shared include compilers, window systems, runtime libraries, database systems and so on.

## Segmentation:

Paging does not presewe users view of a program.

user ~~uses~~ views a program as a collection of functions

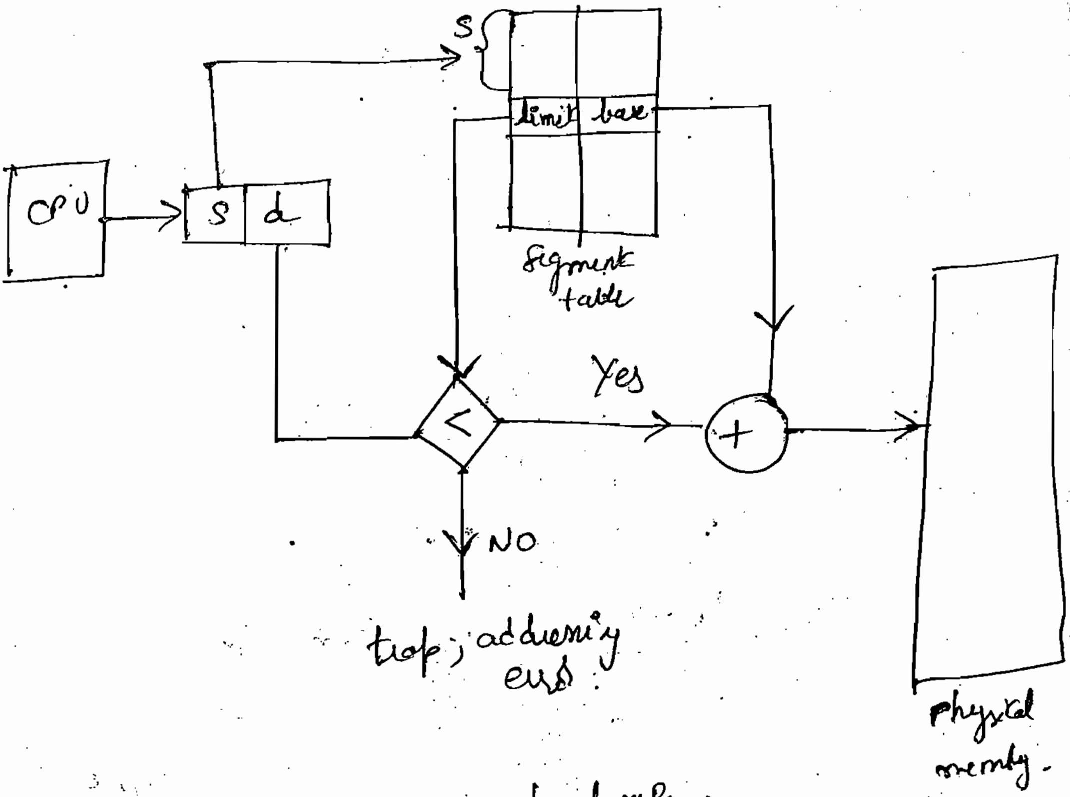


Segmentation is a memory management scheme that supports this user view of memory.

A logical-address space is a collection of segments according to segmentation.

Thus logical address consists of two tuple:

<segment-number, offset>



Segmentation hardware.

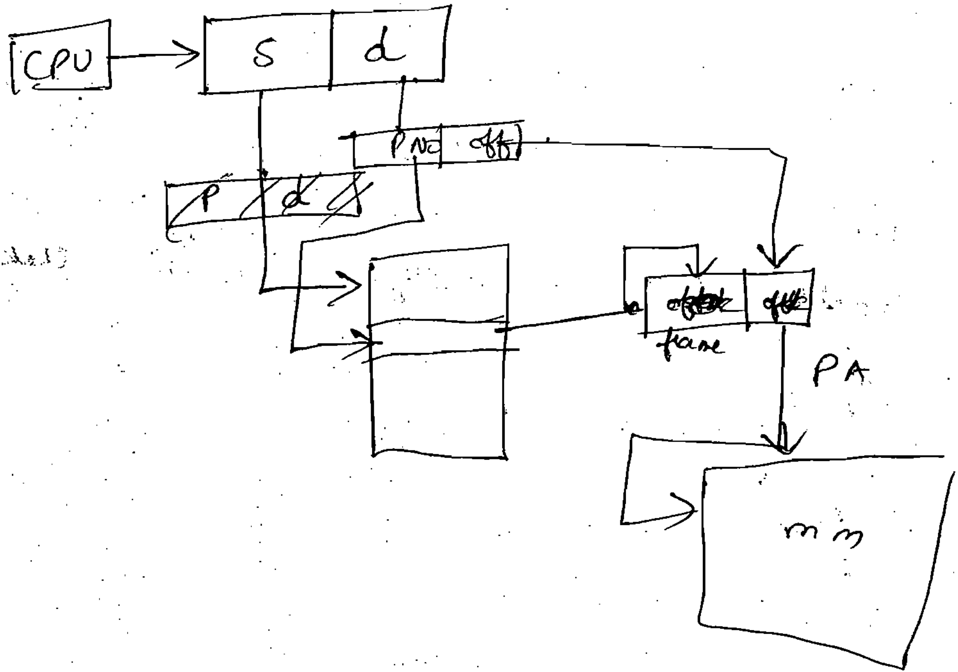
**Disadvantage of sementation:**

→ Like in ~~variable length~~ dynamic size partitioning, pure segmentation suffers from external fragmentation.

→ This problem arises because, we wanted a segment to be contiguously allocated memory

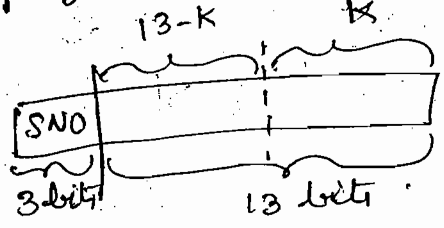
→ The solution to the above problem is to page ~~to~~ divide each segment into pages and memory into frames. This is called

→ The segmentation with paging.



Consider a system with segmented paging supporting LAS = PAS = $2^{16}$ bytes. The LAS is divided into 8 equal size non overlapping segments. The segment is divided into pages and page tables of segments are kept in memory. The page table entry size is 2 bytes. What must be the page size of segment so that the page table of a segment exactly fits in 1 page. (assume that page size is a power of 2).

Sol:



Let page size = $2^K$ B

Page table size = no of pages × size of each entry

$= 2^{13-K} \times 2^1 = 2^{14-K}$ Bytes

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

we need to fit a page table in one page.

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

$$\Rightarrow \frac{2^{14-K}}{2^{K}} = 1$$

$$\Rightarrow 2^{14-2K} = 2^{0}$$

$$\Rightarrow 2K = 14 \Rightarrow K = 7.$$

$$\therefore \text{page size} = 2^{K} B$$
$$= 128 B.$$

## Virtual memory :-

→ virtual memory gives illustration to the programmer that a huge amount of memory is available at his disposal for writing programs larger than the size of available physical memory.

> virtual memory is commonly implemented by demand paging.

→ Demand paging is ~~rather than~~ similar to rather than loading an entire process, into memory, we load only those pages which are needed.

→ A page that is not needed will ~~be~~ remain ~~kept~~ on disk only until it is needed.

## Frame allocation policies :-

Let total frame = 'm'

No of processes = 'n'

Demand of each process 'i' = '$s_i$'

$$\sum s_i = S$$

Let frames allocated for each process = $a_i$.

Frames can be allocated to processes using :

(a) Equal allocation (b) proportionate allocation.

## Equal allocation :-

$\cancel{a_i} = \cancel{\frac{M}{n}}$

allocate frames to all processes equally.

$$a_i = M/n$$

It is useful ~~co~~ only when all process are equal sized.

## Proportionate allocation :-

$$a_i = \frac{S_i}{S} \times M$$

## Page replacement algorithms :-

Reference string: set of successively unique pages referred in the given list of LA's, is called reference string.

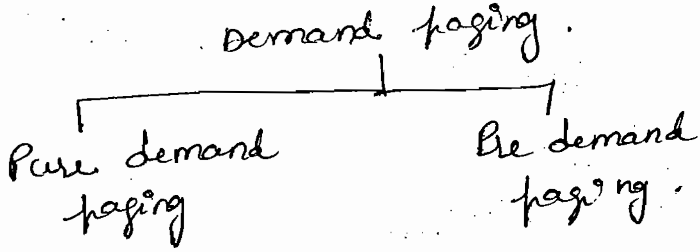
~~Ex~~

Ex: Let size of a page be 100 words, and a process generates the LA's as follows:
122, 142, 148, 150, 172, 112, 754, 787, 756, 896, 012, 064, 074, 158, 172,
666

now reference string = (1, 7, 8, 0, 1, 6)

length of ref string = 6.

unique pages referred = n = 5.



Pure demand paging:

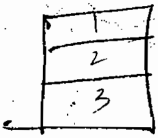
we start with empty frames

Pre demand paging:

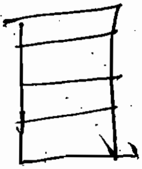
we start with frames filled.

FIFO: Refs 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

3)



P.F = 9



PF = 10.

5F - 5 6F - 5 7F - 5.

Ref: 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5

2) 3 frames = 10   4 frames = 11

~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ ~~4~~ ~~3~~ 5 2 1

Beladys anomaly ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ ~~5~~ ~~4~~ 3 2 1 5

Ref: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2,

1) 0, 1, 7, 0, 1.

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

3 frames = 15 ~~for~~ faults

4 frames = 10 faults

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

→ For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases.

→ This most unexpected result is known as Belady's anomaly.

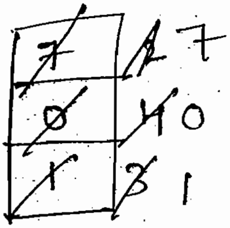
ptimal page replacement :-

→ To avoid Belady's anomaly, optimal page replacement algorithm was proposed.

→ ~~The~~ optimal page replacement is to replace a page that will not be used for the longest period of time.

= 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

R = 1, 0, 7, 1, 0, 2, 1, 2, - - - -



3 frames - 9 PF.

4 frames -

= 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. $S^{R_2}$

3 frames -

4 frames -

5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5 $S^{R_2}$

3 frames

4 frames

-> optimal page replacement algorithm is difficult to implement, because we need to know the future ~~knowledge~~ values of reference string to take current decision.

-> Since no other page replacement algorithm can outperform optimal page replacement algorithm, this is ~~used as~~ mainly used for comparison studies.

## LRU Page replacement:

LRU page replacement algo will replace the page that has not been used for the longest period of time.

S: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

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

3 frames = 12 PFs.

4 frames = ?

$S^R$ = ?

LRU Implementation

a) counters: we maintain a register called clock register which is incremented for every memory ~~page~~ reference.

we maintain a time of use field in every entry of page table.

with every ~~page reference~~ memory reference, the value of the clock register is copied to the time of use field in the page table.

b) stack: we maintain a stack of page numbers.

whenever a page is referenced, it is removed from the stack and puts on the top.

In this way, the top of stack is ~~the~~ always the most recently used page and the bottom of is LRU page.

This stack is implemented as a double linked list.

## Stack algorithms :-

→ A stack page replacement algorithm is called a stack algorithm, if it can be shown that the set of pages in memory for 'n' frames is always a subset of the set of pages ~~in~~ that would be in memory with n+1 frames.

→ Stack algorithms never exhibit Belady's anamoly.

## LRU approximation page replacement :-

### Reference bit :-

Each entry in page table will have a reference bit, associated with every page.

The reference bit ~~is~~ for a page is set, whenever the page is referenced.

### Additional-reference-bits algorithm :

→ we keep an 8-bit byte for each page in a table in memory.

→ At regular intervals, OS ~~will~~ shifts the reference bit for each page into high-order bit of its 8-bit ~~shift register~~ byte, shifting the other bits right 1-bit, discarding the low-order bit.

→ These 8-bit shift registers contain the history of page use for the last eight time periods.

Ex: 11000100 → most recently used among the two.
01110111

Second chance algorithm

→ It is a FIFO replacement algorithm.

→ when a page has ~~to~~ been selected for replacement, we inspect its reference bit. If the value is 0, we proceed to replace this page. If it is 1, we give that page a second chance and move on to select the next FIFO page.

when a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time.

## Enhanced second chance

we use both reference and modified bits as an ordered pair.

(0,0) neither rec — best for replace

(0,1) — not quite good.

(1,0) — it probably be used again.

(1,1) — it probably will be used again soon

## Counting-based page replacement!

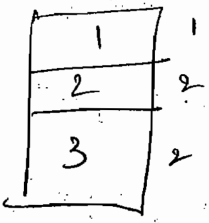
we could keep a counter of the number of references that have been made to each page. we develop the following two schemes.

Least frequently used page replacement algo :-

This requires that pages with smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count

Disadvantages

1, 2, 3, 2, 3, 4, 1, 4, 1, 4, 1 ------



A page which is used heavily during the initial phase of a process, but then is never used again will always ~~stay~~ remain in memory. The solution is to right shift the count by 1 bit at regular intervals.

→ most frequently used (MFU) Page-replacement algo:-

This is based on argument that the page with the smallest count was probably just brought in and has yet to be used.

→ Both MFU and LFU implementations are expensive and they do no approximate OPT replacement well.

## Trashing:

$\underbrace{1, 2, 3, 4}, \underbrace{1, 2, 3, 4}, \ldots$

→ If no of frames allocated ~~are~~ is less than the number of pages in active use, i ~~[illegible]~~ ~~[illegible]~~ Replace/page faults occur very often.

→ Page faults will have significant effect on the performance of a computer system.

→ If there are no page faults, ~~effective~~ ~~eff~~ memory access time = main memory access time.

→ If there is page faults, ~~effective~~ memory accesstime = Time taken to first read the relevant page from disk, and then access the desired word.

→ Let us call ~~this~~ the time taken to process page fault ~~service~~ ~~time of~~ as page fault service time.

Let 'p' be the probability of page fault. ~~Rates~~

Then

effective memory access time

$$= (1-p) \times ma + p \times \text{page fault service time}.$$

If ~~no~~ page fault service time = 25 milliseconds.

and memory access time = 100 nano seconds

Then effective access time in nano seconds

$$= (1-p) \times 100 + p\,(25 \text{ milliseconds}).$$

$$= (1-p) \times 100 + p \times 25{,}000{,}000$$

$$= 100 + 24{,}999{,}900 \times p.$$

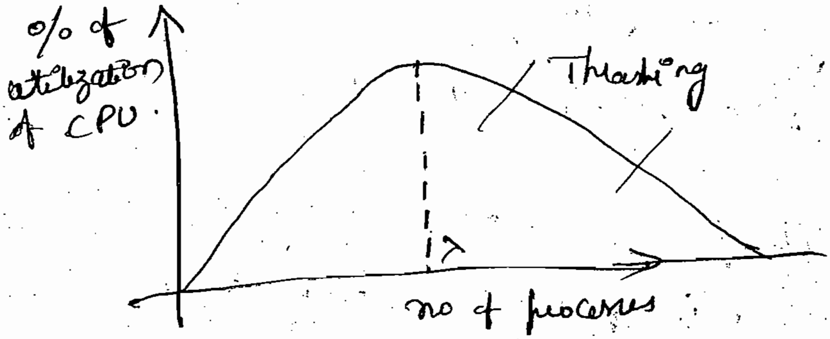
∴ ~~Ememory~~ Effective memory access time $\propto p$.

> A high paging activity is called thrashing.

A process is thrashing, if it is spending more time ~~in~~ paging than executing.

Reasons for trashing :-

1) Lack of memory.
2) degree of multiprogramming is high.
3) Page replacement technique.
4) Too small Page size.
5) Program structure.



when ~~%~~ utilization of CPU is ~~too~~ decreases because of page faults, ~~thus~~ which is because of allocation of less frames to each process.

To ~~is~~ allot more frames to each process, ~~so~~ decrease the number of processes in main memory.

→ If trashing occurs, then ~~deallocate~~ climax the degree of multiprogramming to 'λ'.

→ Program Structure :-

Assume pages are 128 words in size. Consider a java program whose function is to initialize to '0' each element of a 128 by 128 array. The following code is typical:

```
int A[][] = new int[128][128];

for (int j=0; j<A.length; j++)
    for (int i=0; i<A.length; i++)
        A[i][j] = 0;
```

Notice that an array is stored in row major order.

i.e.

A[0][0] A[0][1] - - - - A[0][127], A[1][0],
A[1][1] - - - - A[127][127].

Let size of each ~~a~~ element of array be one [illegible] then ~~the~~ each row will be present in one page.

A[0][0] - - - - - - - - A[0][127]

A[1][0] - - - - - - - - A[1][127]

A[2][0] - - - - - - - - A[2][127].

⋮ ⋮

A[127][0] - - - - - - - A[127][127].

If OS allocates less than 128 frames to this process, then each page reference will produce a page fault. So no of page faults
= 128 × 128 = 16,384 faults.

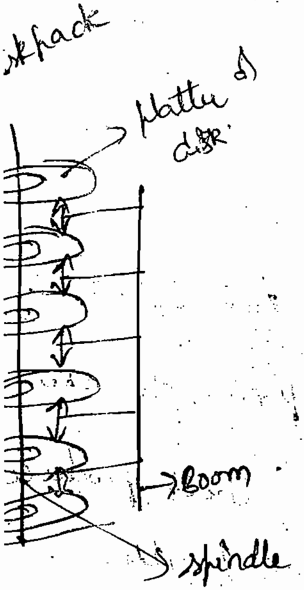
now change the code ~~to~~ as follows:

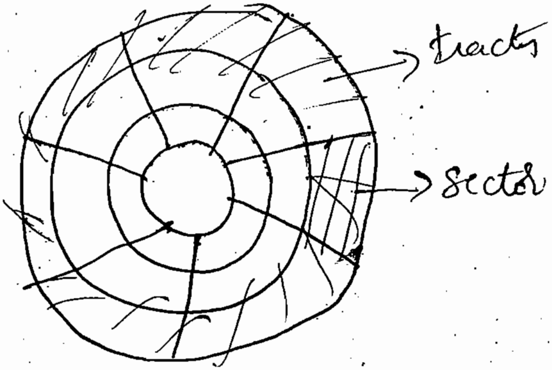
```
int A[][] = new int[128][128];
for(int i=0; i< A.length ; i++)
    for (int j=0 ; j< A.length ; j++)
        A[i][j] = 0;
```

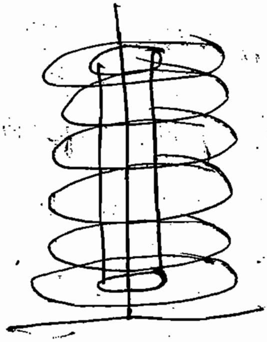
This produces only 128 page faults.

~~Start~~

Disk :-







Capacity = No of ~~tracks~~ surfaces / disk × No of tracks/surface × no of sectors/track × No of bytes/sector

→ Given that no of surfaces = 32, ~~sector~~

no of ~~[illegible]~~ tracks/surface = 64

No of sectors / track = 128.

Sector size = 4096 Byte.

what is the capacity of disk.

→ writing on disk is called recording. There are two types of recording

(a) variable recording density

(b) Fixed recording density.

→ Density of ~~a disk~~ of a track is number of bytes stored per unit distance.

Fixed recording density:

→ ~~In~~ Fixed recording density, the no of bytes/unit distance is constant and so is the distance b/w the bits in all tracks and so ~~no~~ the no of bits/track will vary and so ~~inner~~ ~~track capacity~~ & ~~less~~ the capacity of inner track is less than that of outer track.

→ Linear velocity is used to read/write from disk.

> In variable recording density, the no of bits per unit distance is not constant.

> The capacity of every track is same.

> So the bits in the inner tracks are densely packed and the bits in outer track are sparsely packed.

> Angular velocity is used to read / write

Seek time : (Tseek)

Seek time is the time taken for the disk arm to move the heads to the cylinder containing the desired sector.

~~The Rotational~~

Rotational latency : It is the ~~Rotational lator~~ additional time waiting for the disk to rotate the desired sector to the disk head.

Transfer time : The time taken to transfer a disk block or sector is called transfer time.

→ Disk access time = Seektime + Rotational latency + data transfer time.

→ Rotational latency can be known if the current position of the ~~hea~~ R/w head is given, else, we have to take rotational

→ ~~Stor~~ ~~st~~ latency as ½(time taken for one rotation).

→ ~~To have effi~~

→ To increase the efficiency, many disks are stacked together and it is called hard disk.

→ ~~If each sector is 1KB. and ø~~

→ If we have a file of 10KB, and sector size is 1KB then, instead of storing ~~each~~ 10KB on ~~one~~ a single track, we will distribute it across the disks by a concept called cylinder. ~~If we have~~

→ ~~tea~~

> ~~A disk system of 2GB~~

> Disk formatting :-

each sector of should contain some control information like sector number etc. Such information should be not accessable by users and that space should not be used for storing user information. This is called formatting.

> A disk system of 2GB is constructed with each disk having the following specifications:

(i) It will have 64 tracks/surface

(ii) It will have ~~100~~ 1024 sectors/track.

(iii) It will have 1024 Bytes/sector and the disk is moving at an angular velocity 6000 rpm.

The formatting overhead is 128 bytes/sector

Inner track capacity is 0.7cm. Seek time = 10 ms.

Q:(i) what is the data transfer rate of each disk

(ii) what is the capacity of formatted disk.

(iii). what is the average access time

(iv) If the disk speed is doubled, what is the % of reduction in average access time.

(v) what is the maximum recording density employed in the above recording.

(vi) If each disk is having the data on both surfaces except the I and last one, for which one surface is used as a guard surface. what will be the no of disks needed for the desired capacity.

Sol:)

(i) No of bytes transferred in 1 second.

In one complete revolution

In one rotation, it will cover 1 track.

Given speed is 6000 rpm.

1 min — 6000 rotations.

60 sec.

1 rotation = $\frac{1}{\cancel{6000}_{100}}$

1 rotation — 1 ms.

Capacity of each track = 1024 × 1024

= 1204 KB.

In 10 ms – 1 mB.

1 sec – 100 mB.

∴ data transfer rate = 100 mBPS.

ii) Capacity of a disk = 2 × 64 × 1024 × (1024 KB) bytes.

iii). $T_{ave} = 10\text{ms} + \frac{1}{2}(10\text{ms}) + \frac{1024}{100\text{mBB}}$

= 15.01 m.sec.

iv) If speed of disk is double

$T_{ave_1} \cong 15$ ms

$T_{ave_2} \cong 12.5$ ms

w.r.t 15 ms – 2.5 ms

100 ms – $\frac{2.5}{15} \times 100 = 16.67$.

(v) Inner track diameter = 0.7 cm

Circumference = $2\pi r$

$= \pi(2r)$.

$= \pi D$.

No of ~~bits~~ Bytes in $\pi(D)$ length = 1024 KB.

$\Rightarrow \frac{22}{7} \times 0.7$ cm = 1 MB.

$\Rightarrow$ 2.2 cm $\cong$ 1 MB.

$\Rightarrow$ 1 cm $- \frac{1}{2.2}$ MB/cm.

(vi) ~~(2n-2)/X~~

Let n be the no of disks.

$(2n-2) \times 2^{26} = 2^{31}$.

$\Rightarrow 2n-2 = 32$.

$\Rightarrow n = 17$.

→ A disk has 8 equidistance tracks, the diameter of inner track is 1 cm, while outermost track is 8 cm. The inner most track has a storage capacity of 10 MB. What is the amount of data that can be stored on disk if it is used with a drive that rotates (i) with constant linear velocity
(ii) Constant angular velocity.

a) 80 MB, 2040 MB   b) 2040 MB, 80 MB
c) 80 MB, 360 MB   [✓ d)] 360 MB, 80 MB.

In linear 10+20+ . . . 80 = 360 MB.

In angular, 10+10+ . . . 10 = 80 MB.

## Resource-request algorithm

1. If $Request_i \leq Need_i$, go to the step 2. otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. otherwise $P_i$ must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to $P_i$ by modifying the state as follows:

    $Available = Available - Request_i$;

    ~~Avai~~ $Allocation = Allocation + Request_i$;

    $Need_i = Need_i - Request_i$;

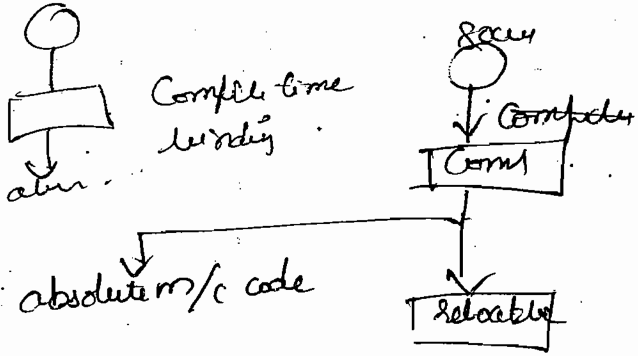
If the resulting resource allocation state is safe, the transaction is completed and process $P_i$ is allocated its resources.

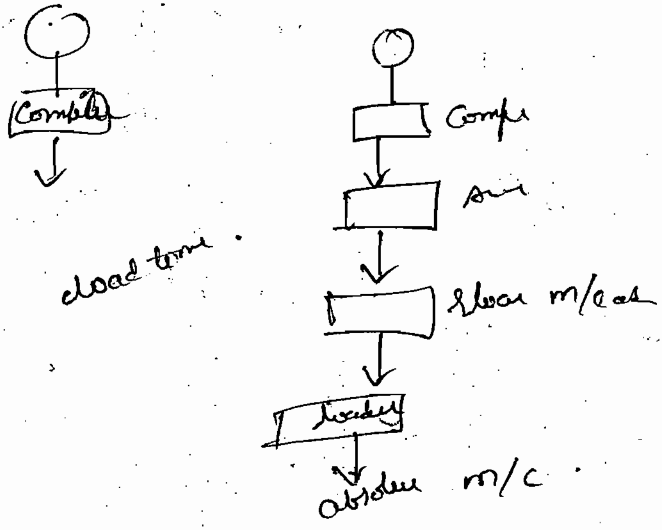
## Safety algorithm

1. Let "work" and "finish" be vectors of length 'm' and 'n', respectively. Initialize work = Available and finish [i] = false for i = 1, 2, --- n.

```
2. Find an i such that both
       a. Finish [i] = false
       b. $Need_i \le work$.
   if no such i exists, go to step 4.
3. work = work + $allocation_i$
   Finish [i] = true
   goto step 2.
4. If Finish [i] = true for all i, then the system
   is in a safe state.
```

This algorithm requires an order of $mn^2$


Compile time
binding
abs.
Source
Compiler
Comp
absolute m/c code
Relocatable


Compiler
load time
Comp
Asm
Reloc m/code
loader
absolute m/c

~~A B~~

|  | ~~Execution~~ Allocation |  |  |  | available |  |  |  | maximum requirement |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | A | B | C | D | A | B | C | D |  |  |  |  |
| $P_1$ | 1 | 2 | 2 | 1 | 3 | 1 | 1 | 2 | 3 | 3 | 2 | 2 |
| $P_2$ | 1 | 0 | 3 | 3 |  |  |  |  | 1 | 2 | 3 | 4 |
| $P_3$ | 1 | 1 | 1 | 0 |  |  |  |  | 1 | 1 | 5 | 0 |

---

need

| | | | | |
|---|---|---|---|---|
| $P_1$ | 2 | 1 | 0 | 1 |
| $P_2$ | 0 | 2 | 0 | 1 |
| $P_3$ | 0 | 0 | 4 | 0 |

$\langle P_1, P_2, P_3 \rangle$ ✓

safe state.

At the point $P_2$ asks for 1 B.

request = 0 1 0 0.

~~Re~~available: 3 0 1 2.

unsafe

~~Request~~ = $P_3$ asks for (0, 0, 2, 0)

allocation cannot be met.

$P_3$ asks for (0, 0, 1, 0)

$\langle P_1, P_2, P_3 \rangle$ safe sequence.

[illegible] [illegible] [illegible] — 7

[illegible] [illegible] [illegible] — 5

[illegible] [illegible] [illegible] — 6

12 ③ ✓

Current Needs

| | | |
|---|---|---|
| 10 | 5 ✓ | ~~③ ✓ -2 = 1~~ |
| 4 | 2 | |
| 9 | ~~7~~ | 5 |

(P₁) ⑤

~~②~~ ✓

< P₁, P₀, P₂ >

| | max demand | allocation | Need | |
|---|---|---|---|---|
| $P_0$ | 10 | 5 | 5 | |
| $P_1$ | 4 | ~~3~~ 2 | 2 ✓ | ~~2~~ 4 |
| $P_2$ | 9 | 3 | 7 | |

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

(45)

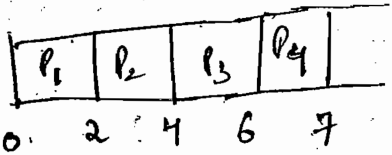
$$= \alpha t_n + (1-\alpha)\left(\alpha t_{n-1} + \alpha(1-\alpha)\cancel{t}\,\tau_{n-1}\right)$$

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

~~P1~~ ~~P2~~ ~~P3~~ P4 ~~P5~~ ~~P6~~.

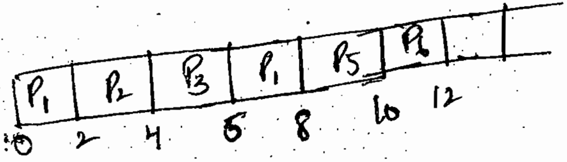
~~P1 P2 P3 P1~~

~~P1~~ ~~P2~~ ~~P3~~ ~~P4~~ P5 P6 P1 P2 P3



~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P5~~ ~~P5~~ ~~P6~~ P3 P5 P6

| | AT | ~~ST~~ | BT |
|---|---|---|---|
| 1 | 0 | ~~4~~ ~~2~~ | ~~4~~ ~~2~~ 0 |
| 2 | 1 | ~~5~~ ~~3~~ | ~~5~~ 3 |
| 3 | 2 | ~~3~~ ~~1~~ | ~~3~~ 1 |
| 4 | 3 | ~~1~~ ~~0~~ | 1~~0~~ |
| 5 | 4 | ~~6~~ | ~~6~~ 4 |
| 6 | 5 | ~~5~~ | ~~5~~ 3 |



OSI

application
↓
Presentation } user oriented (Changes based on application
↓
session
↓
Transport ← Interface b/w groups.
↓
Network layer } N/W oriented (Changes based on N/W)
↓
Data link
↓
physical layer

optional

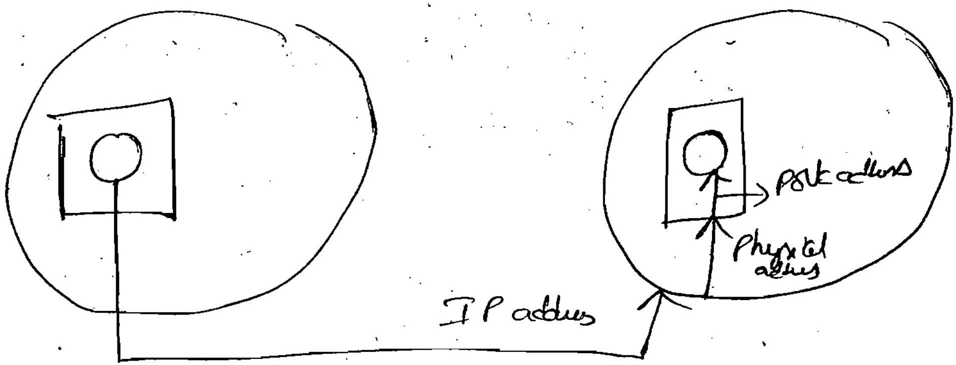
session → Thin layer

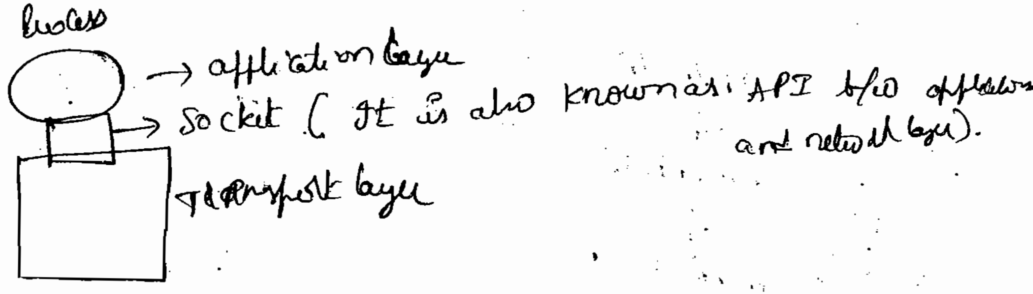
Transport → Thick layer

Network layer → Complicated layer

S/W ↓ H/W

physical layer: Pure H/W layer





## Socket programming with TCP

Any N/w application Consists of a pair of programs, a client program and a server program — residing in two different hosts.

when these two programs are ~~Created~~ executed, a client and a server process are Created and these po two processes communicate with each other by reading from and writing to sockets.

→ To write any of n/w program, there will be set of rules called prototype specified by RFC.

→ we can directly implement RFC's.

Ex:- RFC-959 client side of FTP and server side of FTP.

RFC/

network programming..

follow in RFC protocols

Ex: RFC-959 FTP server and client

client can be created by a programmer and server by another programmer

need not follow RFC

Ex: proprietary n/w application

client and server has to be created by same developer of development team.

~~→ Proprietary n/w applications~~

~~Step 1: Decide whether the application is to run on TCP or UDP.~~

~~N/w applications~~

~~Socket programming with TCP~~

~~Socket programming with UDP.~~

Socket programming with TCP:-

~~-> The client initiates the contact with the server..~~

~~-> For the server should always be running.~~

~~-> you can compare this as someone knowing~~
~~(Socket Programming~~

Step 1: (Socket Creation)

Creat a socket, which is done with the following operation:

int socket (int domain, int type, int protocol).

domain: specifies protocol family that is going to be used

PF_INET denotes internet family.

PF_UNIX denotes unix pipe facility.

PF_Packet denotes direct access to network Interface.

Type: Type argument indicates the semantics of the communication.

SOCK_STREAM is used to denote byte stream.

SOCK_DGram is used to denote message oriented served.

Protocol: Protocol argument identified the specific protocol that is going to be used

→ The return value from the socket is a handle for the newly created socket, i.e, an identifier by which we can refer to socket in the future.

**Step 2:** (Connection establishment) This step depends on whether you are a client or a server.

**Server:** on a server machine, the application process performs a passive open — the server says that it is prepared to accept connections, but it does not actually establish a connection. The server does this by invoking the following three operations:

```
int bind (int socket, struct sockaddr * address,
                      int addr_leng)

int listen (int socket, int backlog)

int accept (int socket, struct sockaddr * address,
                        int * addr_len)
```

Bind: the bind operation, as its name suggests, binds the newly created socket to the specified address.

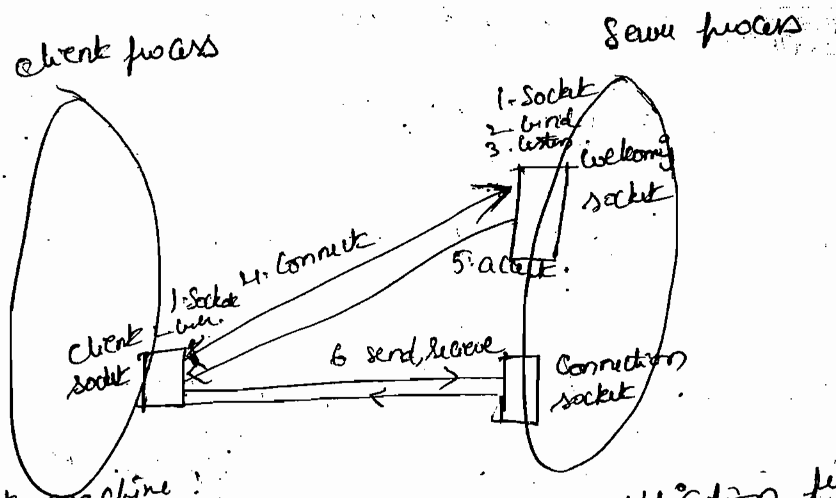
* This is the n/w address of the local participant - the server. When used with internet protocols, "address" is a data structure that includes both IP address of the server and a TCP port number.

web servers - 80

**Listen:** The listen operation then defines how many connections can be pending on specified socket.

-> **Accept:** ~~of~~ This operation carries out the passive open.

It is blocking operation that does not return until a remote participant has established a connection, and when it does complete, it returns a new socket that corresponds to this just established connection, and the address argument contains the remote participants address.



client machine:

→ on the client machines, the application process performs an active open, that is, it says who it wants to communicate with by invoking the following single operation:

```
int connect( int socket, struct sockaddr *address,
                         int addr_len).
```

address contains the remote participants address.

This is a blocking call, and does not return until TCP has successfully established a connection

Step 3.) (Communication)

once a connection is established, the application processes invoke the following two operations to send and receive data

int send( int socket, char * message ; int msg_len
int flags)

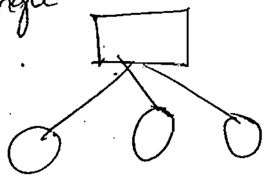
int recv( int sock, char * buff , int buf_len,
int flags).

→ Here flags control details of operation.

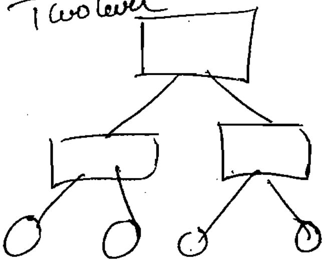
→ close ( int socket ) : used to close a socket.

# File systems

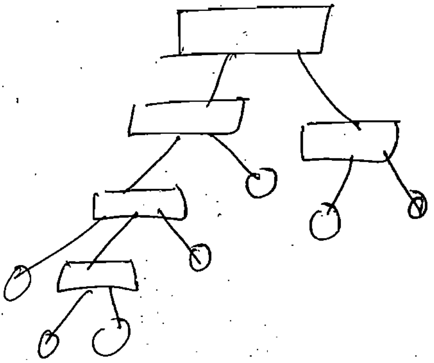
Single level

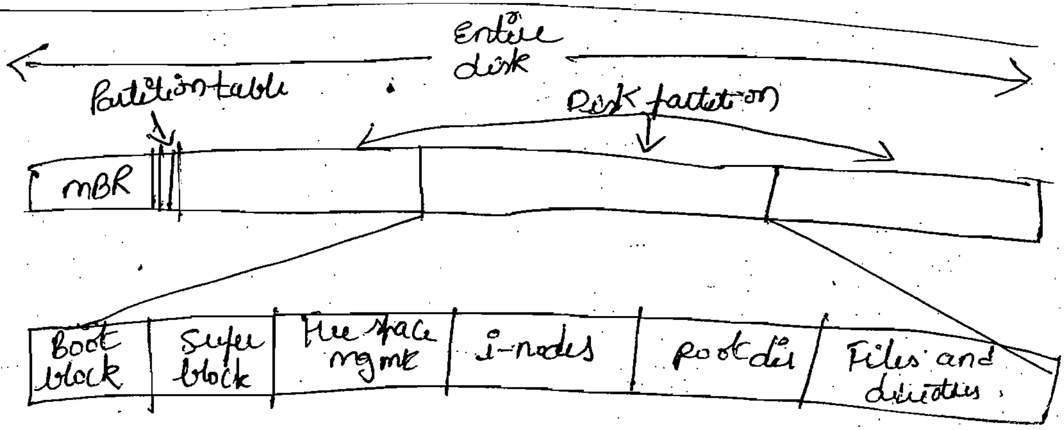


Two level

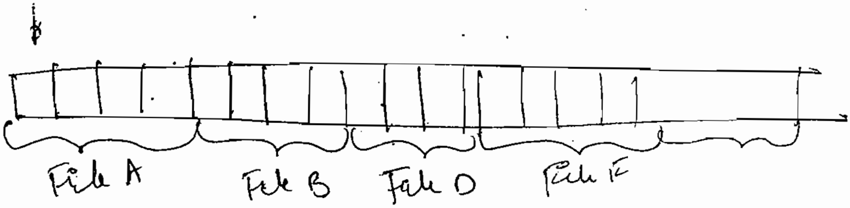


multiple hierarchy



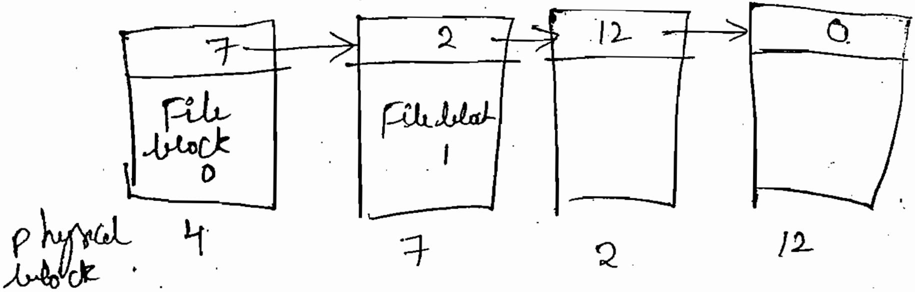


## Contiguous allocation of files

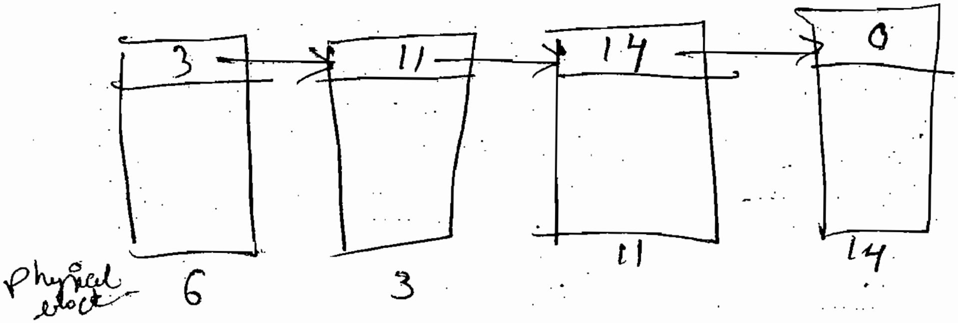


## Storing a file as a linked list of disk blocks

File A.



File B.

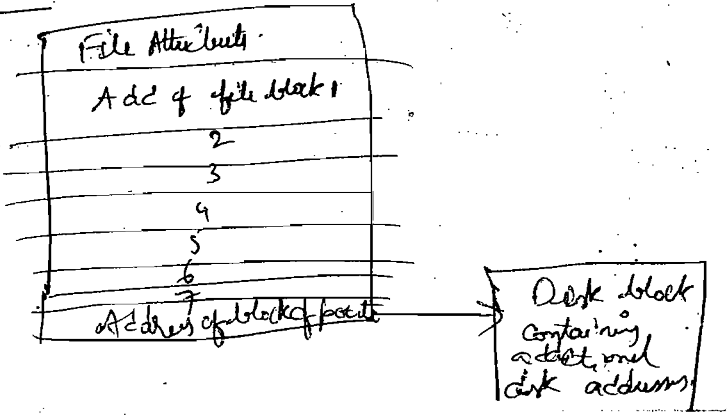


inked [illegible]

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 10 |
| 3 | 11 |
| 4 | 7 |
| 5 | |
| 6 | 3 |
| 7 | 2 |
| 8 | |
| 9 | |
| 10 | 12 |
| 11 | 14 |
| 12 | -1 |
| 13 | |
| 14 | -1 |
| 15 | |

LL allocation using a FAT in main memory

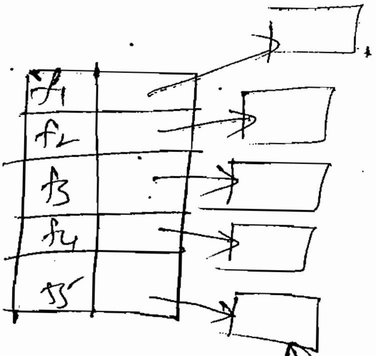
An example i-node



## Implementing directories:

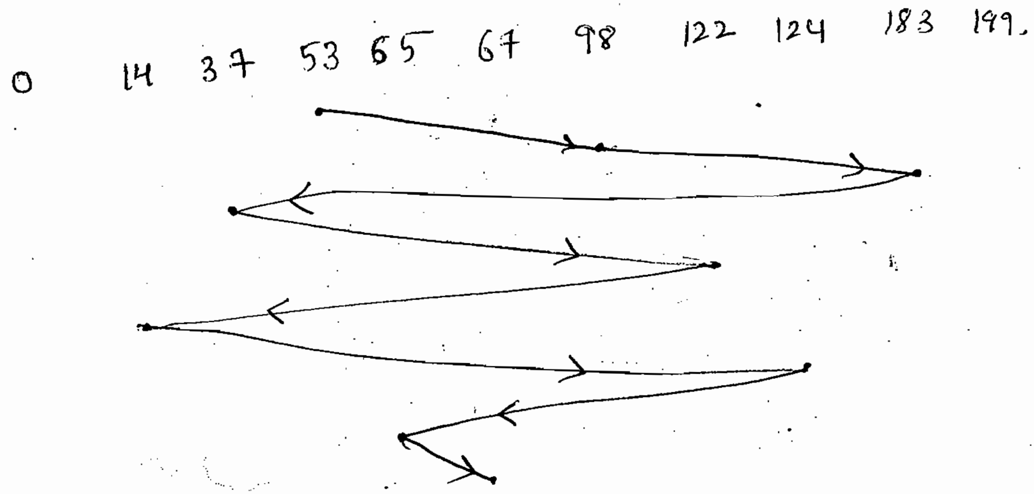
| | |
|---|---|
| f1 | attributes |
| f2 | attributes |
| f3 | attributes |
| f4 | attributes |

a) Fixed size entries with disk address and attributes in directory entry.



data structure containing the attributes.

## FFS



SSTF: