Personal Data

Name :

School /College :

Section/Branch :

Roll No:/Reg.No :

Subject :

Address :

Mobile :

E-Mail :

facebook :

Contact Person Name & No. on Emergency:

REDUCE

Lowering the amount of waste Produced

REUSE

Using materials reapeatedly

RECYCLE

Using materials to make New Products

RECOVERY

Recovering Energy from Waste

LANDFILL

Safe disposal of waste to Landfill

We all need people who will give us feedback. That's how we improve.

-Bill Gates

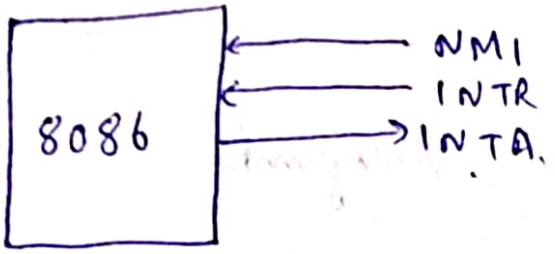
ideas inside

INDEX

NAME: Shreyansh STD: SEC: ROLL NO: SUB: OS

S.No.	Date	Title	Page No.	Teacher's Sign/ Remarks
		Mid Sem - 30 Project - 20 Assignment - 10 End Sem - 40		

Ultima Books



// INT 80H ⇒ user generated interrupt

Interrupt flag

1 — ready to accept the interrupt

0 — it is masked so cant accept the interrupt

// IDTR interrupt decripted table register

↳ starting location of interrupt vector table.

Entries in IDTR is stored by OS // check

- There are three hardware interrupts available on the processor
  1) NMI Non Maskable Interrupt
  2) INTR
  3) INTA

NMI → When it is active, user has no control over it.

INTR → When it is active, user has some control over it.

Due to some internal operation, we can get an interrupt (INTA type interrupt).
  Eg dividing by zero.

ISR ⇒ Interrupt Servie Routing.

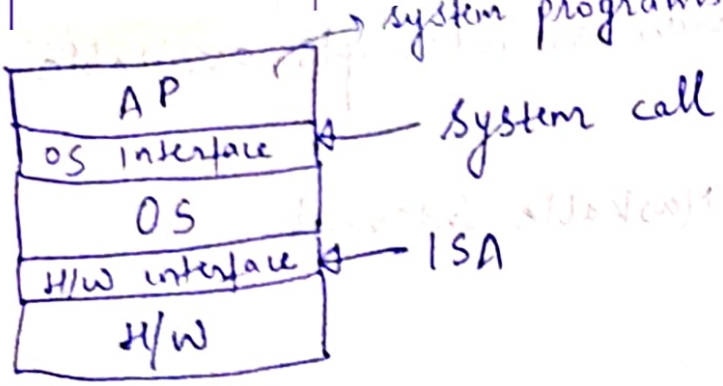
// There are 256 interrupts.

INT 00H
:
INT 255H

Interrupt Vector: hold the address of interrupts

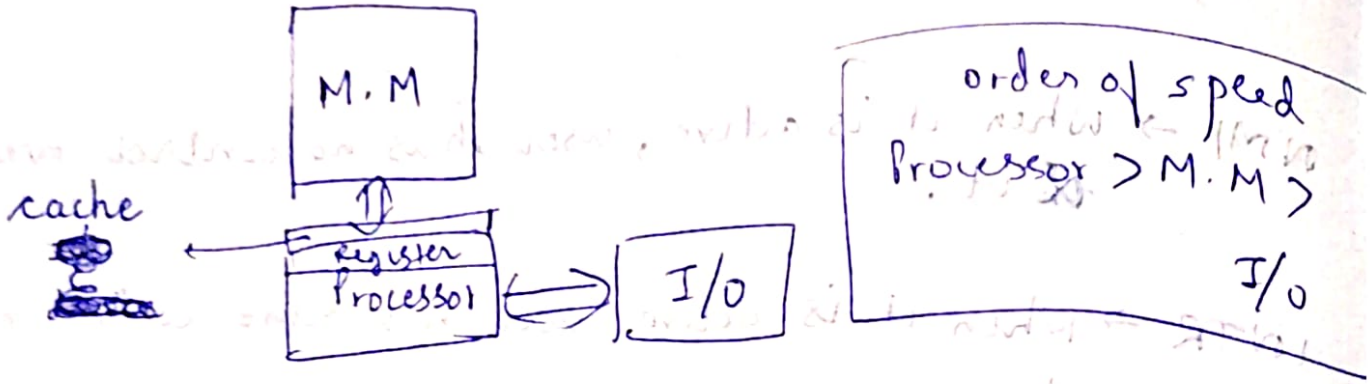
Real Mode :- Vector starts at 0000H

Protected Mode :- ~~[illegible]~~



user level
OS code
user level
OS
user

OS code



* Polling ⇒

```
while(not ready)
{
}
perform I/O.
```

count = read (fd, buffer, n)

push parameters in stack


```
push n
push buffer
push fd
call read()
```

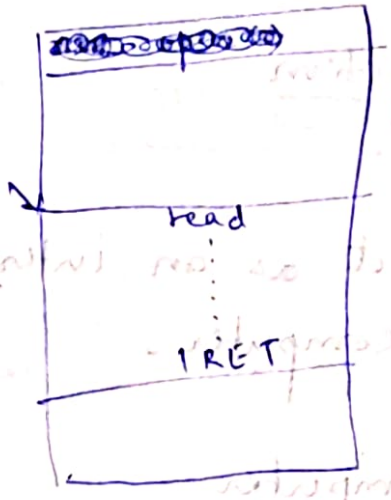
} Application Program

Library

```
load EAX, call no
INT 80H
```

} compiler generates call to the func

OS call



System call no for open is 0.

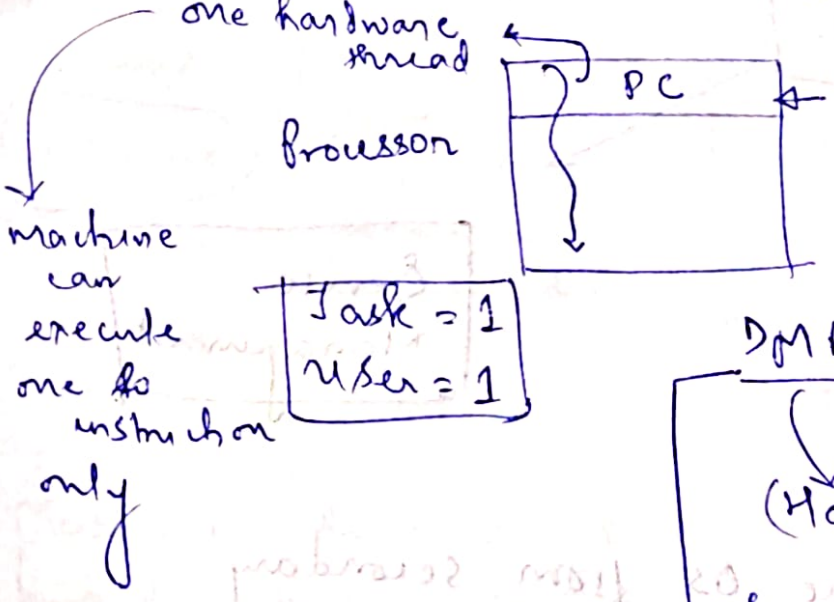
IRET returns the control Then we get RET and operation is terminated.

Process Management System Call :-

fork()
kill()
wait()
signal()

file :- open(), read(), write()

Uniprocessor System (one program counter & one register set)



DMA → direct Memory Access transfer

(Hard disc)

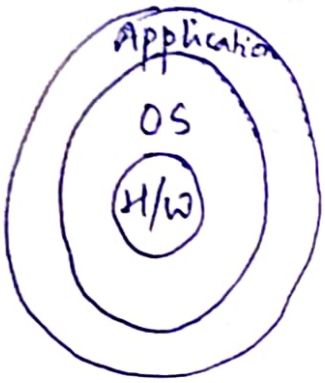
speed of this transfer is very high (advantage)

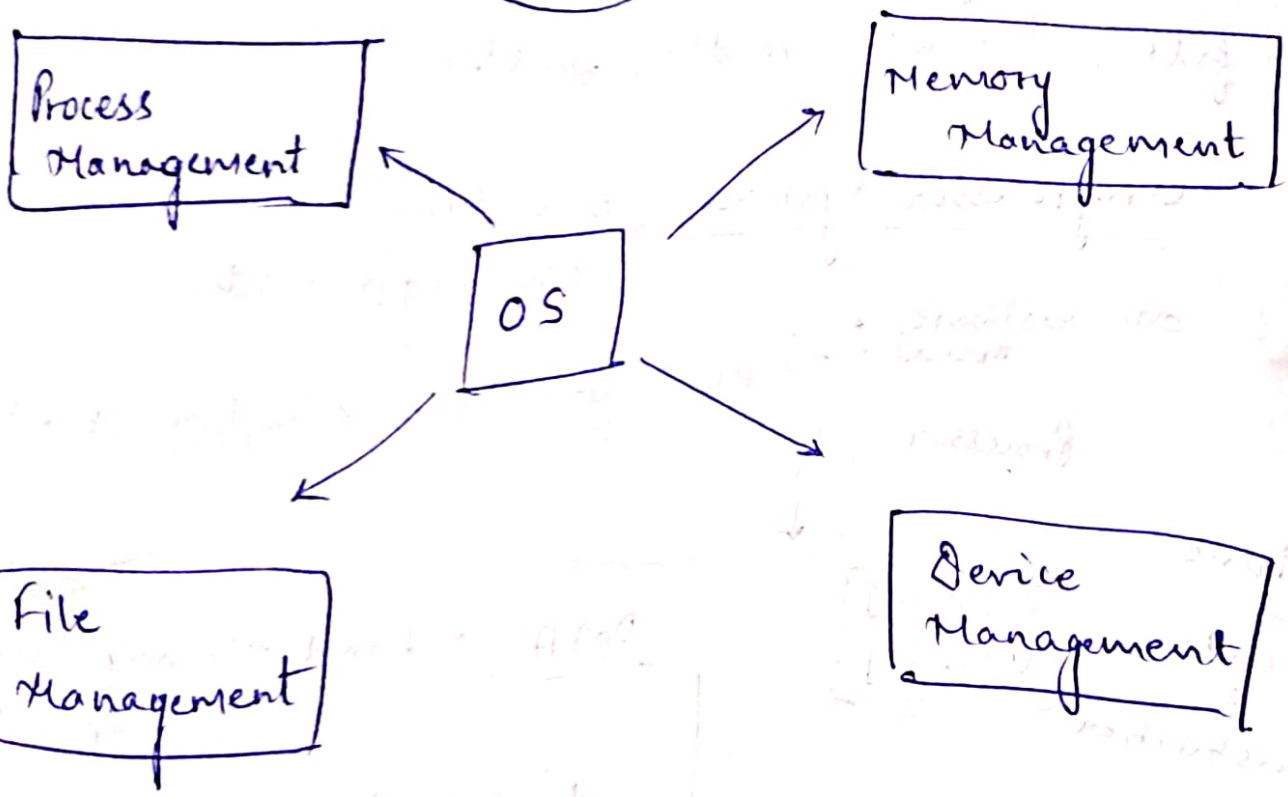
Operating System

OS is a program which act as an interface b/w hardware & User of the computer.

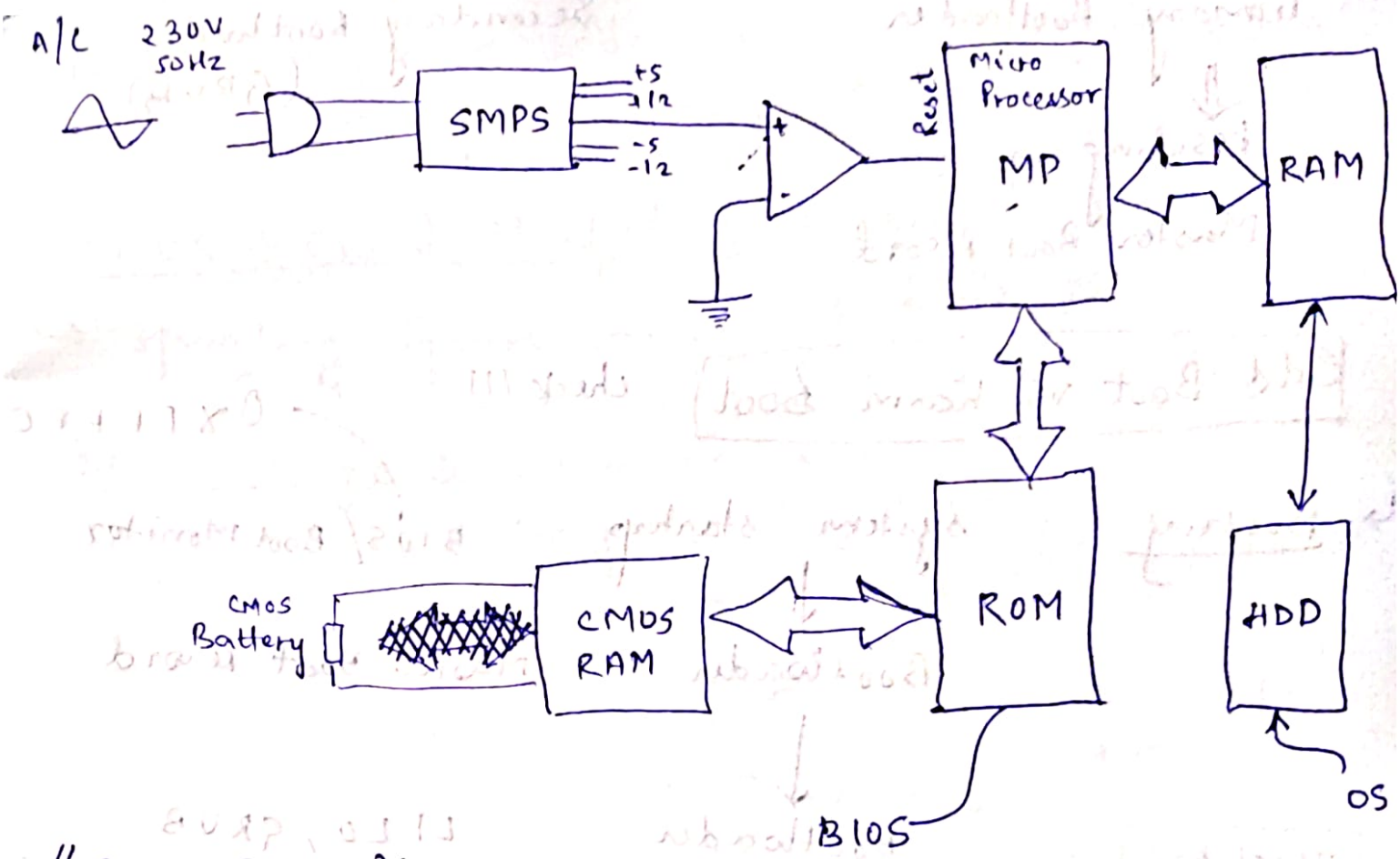
Ring Architecture of Computer

VxWorks, QNX] embedded OS





Booting :- loading the OS from secondary memory to primary memory.



// Power, $P = CV^2f$

C = capacitance
V = voltage
f = frequency

P (1.1)

POST (Power On Self Test)

BIOS

Firmware ⇒ Software which can't be edited.
Eg ROM

CMOS RAM
↓↓
(less power consumptio

Bootable information for CMOS RAM.

Master Boot Record
- Bootstrap loader
- Partition table.

Primary Bootloader

⇩

residing in

Master Boot Record

(GRUB)

Cold Boot VS Warm Boot

check !!!

↳ Booting

system startup

↓

Bootloader

↓

Bootloader

↓

Kernel

↓

Init

0x FFFF0

BIOS / Boot Monitor

Master Boot Record

LILO, GRUB

Linux

/etc

/run

Kernel-Row

⇩

check it.

1) Turn On
2) CPU jump to address of BIOS
3) Bios runs POST (Power On Self Test)
4) Find Bootable devices
5) Load & execute Bootloader from MBR.
6) Load Kernel Boot Loader

Load OS.

BIOS ⇒ software code run by a computer when first powered on.

GRUB ⇒ Grand Unified Bootloader

↳ operating system independent bootloader.

Kernel Image ⇒ not a executable kernel, but a compressed kernel image.

⇒ > 512 kB.

⇒ always stored in memory until turned off.

init process ⇒ first thing the kernel does, is to execute init program.

⇒ Init is the root/parent of all process executing in linux.

grep ⇒ pattern matcher

⇒ id of init is '1'

id 0 ⇒ Scheduler

Linux

file process

mkdir folder
cd folder
folder/ ls -a

output ⇒ . ⇒ for current direc
.. ⇒ for previ
du

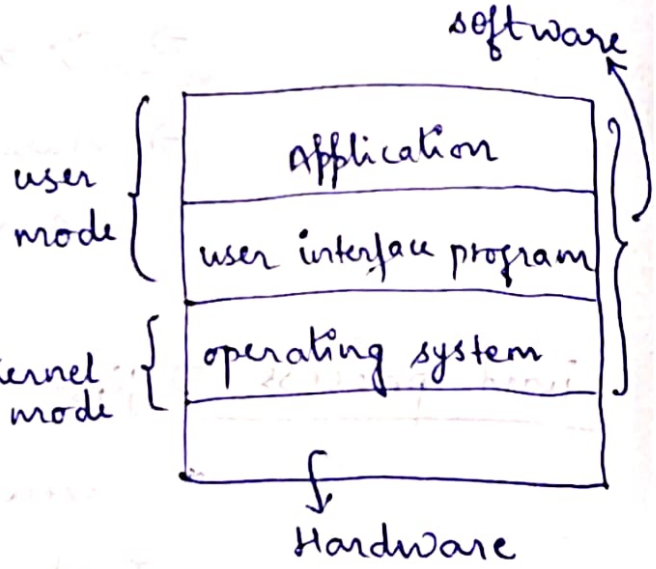
Agenda

14/8

↳ Process
- Process Life Cycle
- Process Context
- Process Switch

Process - is a program in execution

Process context comprise of
↳ Binary program
↳ Data
↳ Process Stack
↳ Process Status Word
↳ other resources

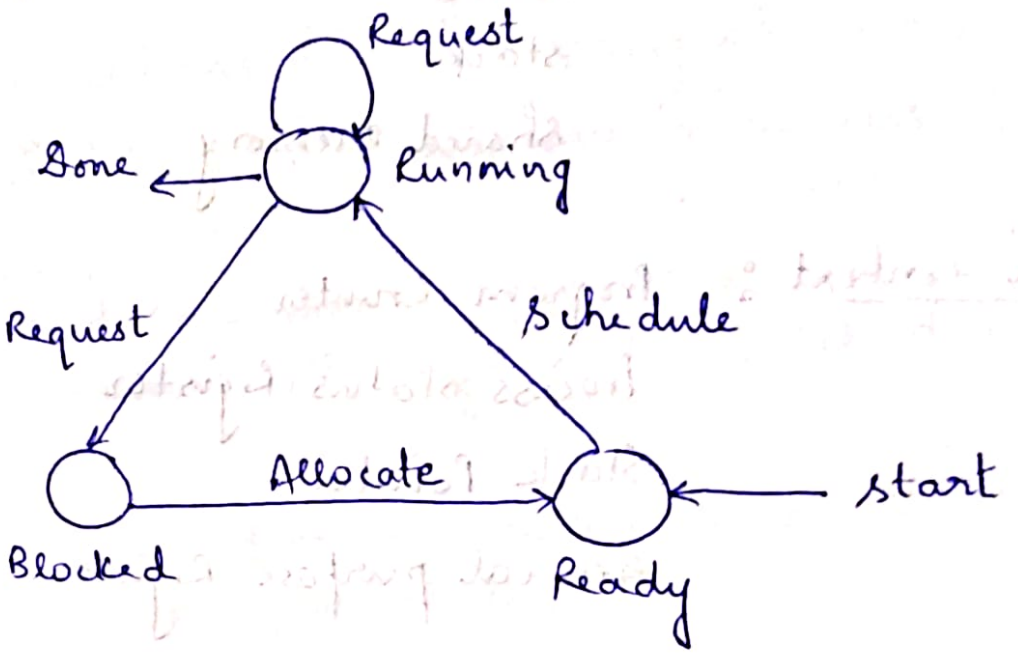


Hardware Process :-
↳ When the computer is powered up, it begins the fetch-execute cycle for the program that is stored at the bootstrap point.

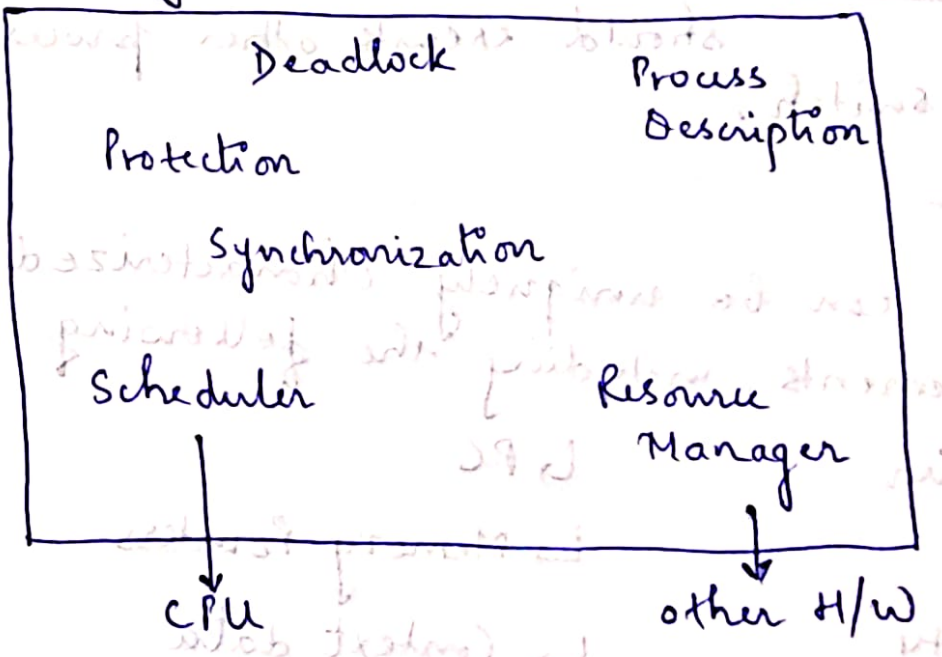
Process consists of text, data & stack.

Process descriptor (table) where OS will keep all the information it needs to manage.

State of a process :



Process Manager



//Multiple processes co-exist.

Process Context → It consists of the contents of user address space, content of hardware registers & kernel data structures. It is the u of user-level-context, register context & sys- level context.

User - level context :- text
data
stack
shared Memory

Register context :- Program Counter
Process status Register
Stack Pointer
General purpose Register

↳ Context switch :- When the kernel decides that it should execute other process, it does a context switch.

Self Notes :-

A process can be uniquely characterized by a no of elements, including the following

↳ identifier
↳ state
↳ Priority
↳ I/O status info
↳ PC
↳ Memory Pointers
↳ Context data
↳ Accounting info

Process Descriptor table :-

e of process Descriptor ⇒

priority of process
whether it is running in the CPU or blocked
what address has been assigned to it.
which files it is allowed to address & so on....

⇒ It is a task_struct type structure whose field contains all information about a process.

↳ There are five associated data structures with process descriptor table.

Process Control Block

Memory Management

// Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but not that take disk addresses. Therefore any instruction in execution, and any data being used by the registers must be in one of these direct access storage devices. If any data are not in memory, they must move there before CPU can operate on them.

// Base Register ⇒ The base register holds the smallest legal physical memory address.

// Limit Register ⇒ The limit register specifies the size of the range.

For eg; if the BP register holds 300040 and the limit register holds 120900, then the program can legally access all addresses from 300040 to 420939.

Protection of memory space is accomplished by having the CPU hardware compare every

address generated in user mode with the registers.

// Any attempt by a program executing in the user mode to access the operating system's memory or other user's programs memory result in fatal error. This scheme prevents a user program from modifying the code / data structures of OS / other user

// The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be accessed only in kernel mode and since only the operating system executes in kernel mode, only the OS can load base and limit registers. This scheme allows the OS to change the value of the registers but prevent the user programs from changing the register contents.

Address Binding :-

Usually a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory & placed within a process. Depending on memory management, the process may be moved b/w disk & memory during execution. An input queue of processes to be

brought to memory for execution is formed.

// The normal procedure is to select one of the process in the input queue, load in memory, execute it. Eventually, the process terminates & its memory space is declared available.

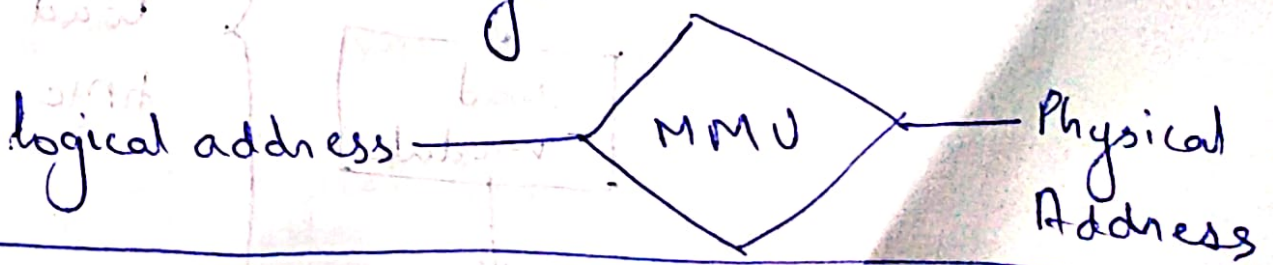
The binding of the instructions and data to memory addresses can be done at any step along the way :-

↳ Compile Time :- If you know at compile time, where the process will reside in memory, then absolute code can be generated.

for eg; if we know that a user process will start at location R, then the generated compiled code will start at that location & extend up from there

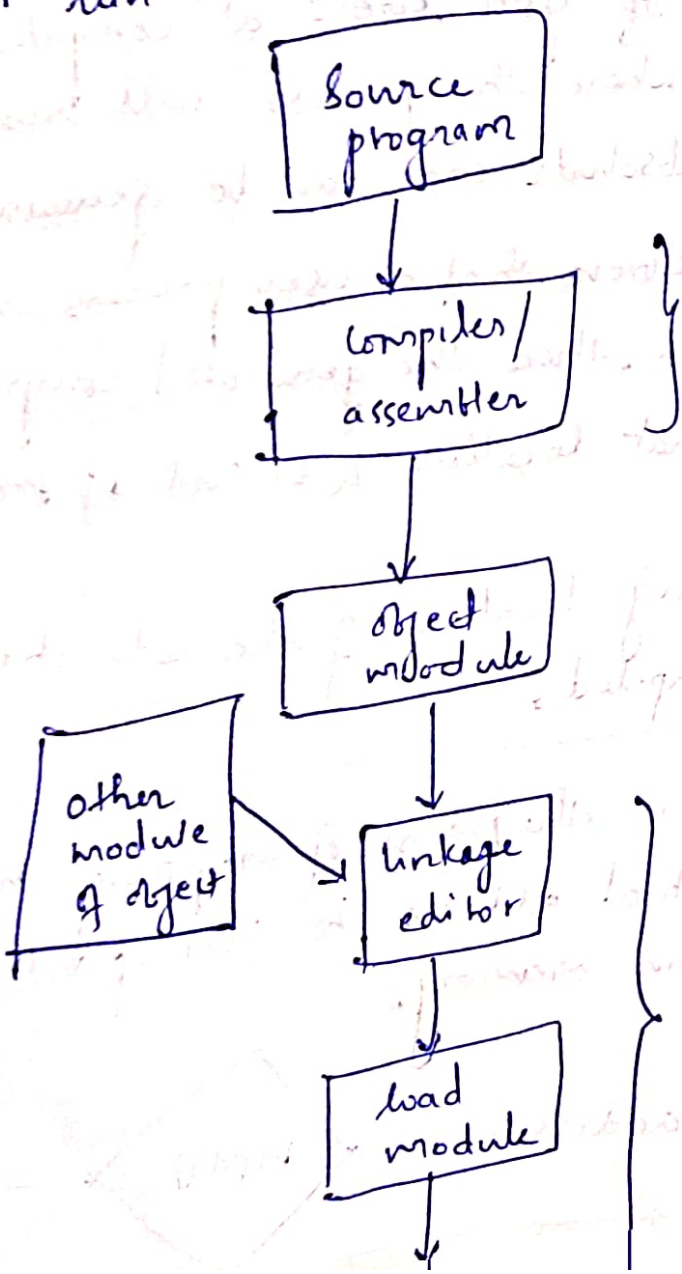
// If the starting location of the code changes, it must be recompiled.

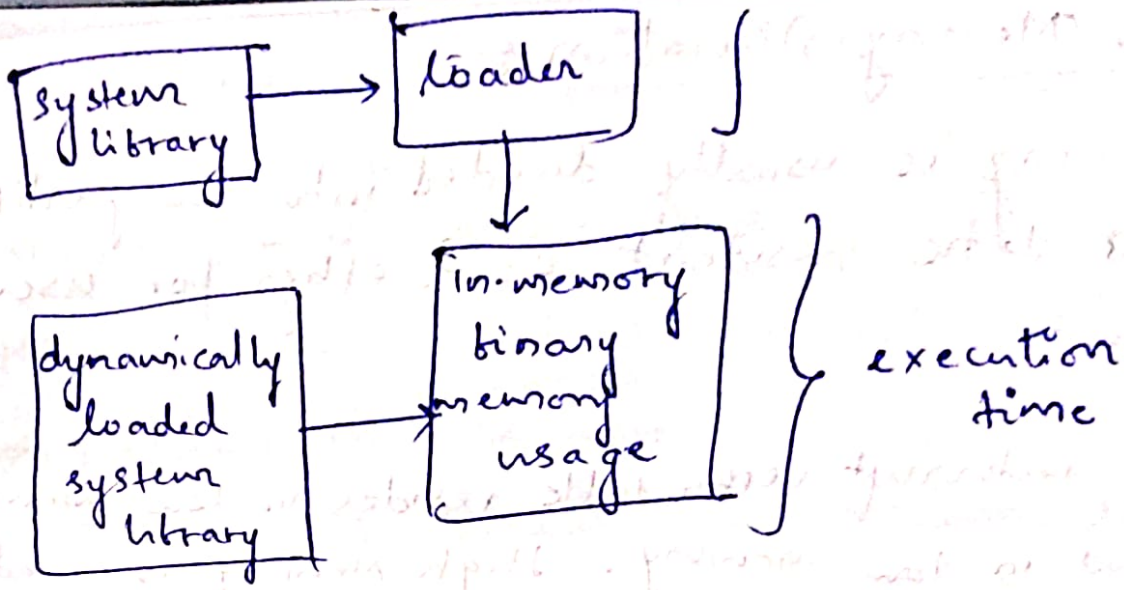
Address binding is the process of mapping the program logical or virtual address to corresponding physical address in main memory.



... o If its not known at compile time, where the process will reside in memory then the compiler must generate relocatable code. Final "binding" is delayed until load time. If the starting address changes, we need only reload the user code.

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





Q) Difference between static & dynamic libraries?

Static libraries, while reusable in multiple programs are locked into a program at compile time whereas dynamic libraries exist as seperate files outside the executable file.

Logical Address vs Physical Address space

Address generated by CPU ⇒ logical address

Address generated (seen) by memory unit ⇒ .

The compile-time & load-time Address binding method generate identical logical & physical address.

Execution time address binding scheme results in different logical & physical address.

{Set of all logical address} ⇒ logical address space

Contiguous Memory Allocation :-

// The memory is usually divided into two partitions one for the resident OS & other for user process.

Since the interrupt vector table resides in low memory OS is put in low memory. High Memory is used for user progsams.

In contiguous memory allocation, each process is contained in a single contiguous section of memo

Memory Allocation & Protection :-

Base Register (Relocation register) & limit registers are used for protection. The relocation registers contains the value of smallest physical address, the limit register contains the range of logical addresses.

Relocation reg value + limit reg value < him reg = Physical Address

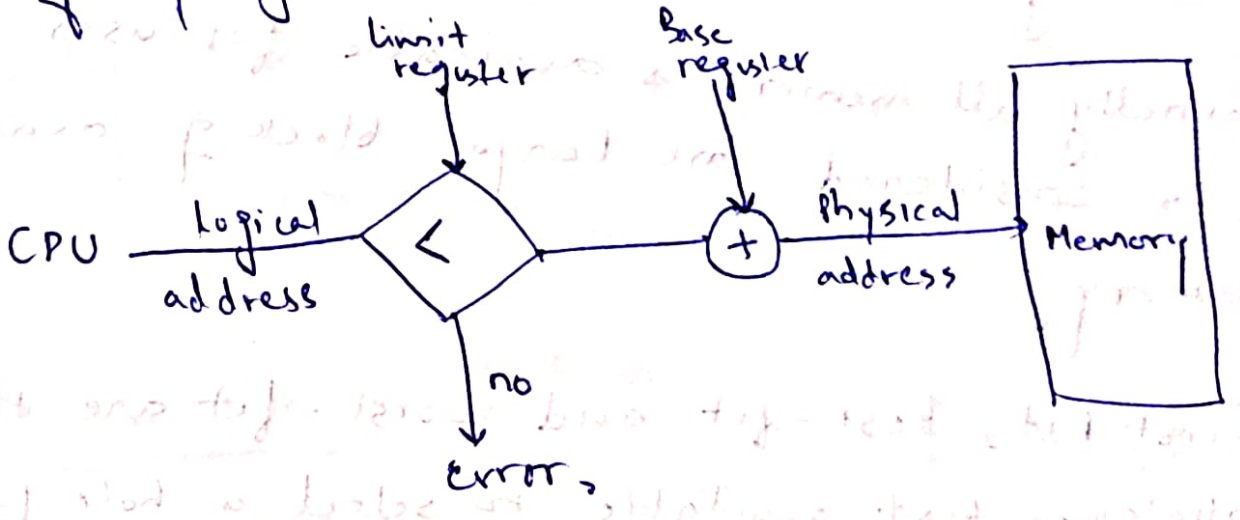
MMU maps the logical address dynamically by adding the value in the relocation register. This mapped

address is sent to memory

When the CPU scheduler selects a process for execution the dispatcher loads the relocation & limit registers with the correct values as part of context switch. Because every address generated by CPU is checked against these registers, we can protect the OS & other user program.

7. // Transient operating system code :-

When needed, the code is loaded in memory and when not needed, it is removed from memory. This allows OS to change its size during program execution.



Memory Allocation :-

One of the simplest methods for allocating memory is to divide into several fixed-sized partitions. Each partition may contain only 1 process.

// Degree of multiprogramming is bound by no of partitions.

When a partition is free, a process is selected fr the input queue & loaded into free partition. When the process terminates, the partition becomes available for another process. This method is genera not used.

Variable partition scheme ⇒ The OS keeps a table indicating which part of the memory are available and which are occupied

* Initially all memory is available for user processes & is considered one large block of available memory.

First Fit, best-fit and worst-fit are the strategies best available to select a hole from set of available holes.

First fit :- First hole which satisfy the memory constraints of the process gets filled.

Best fit :- Allocate the smallest hole that is big enough. It produces smallest leftover hole. (Searching in entire list).

Worst fit :- Allocate the largest hole. Again, we must search the entire list. This strategy provides largest leftover hole.

// Generally first fit & best fit are used.

Fragmentation :-

Both first fit & best fit strategies for memory allocation suffer from external fragmentation.

↳ Fragmentation → internal fragmentation
→ external fragmentation

↳ Internal fragmentation :-

It occurs when we allocate a set of blocks (memory) and not use it fully. It can be corrected by dynamic memory allocation.

process is loaded and removed from mem
the free space (memory) is broken into little
pieces. External fragmentation exists when there
is enough total memory space to satisfy a
request but the available space is not contiguous.

Statistical analysis of first fit reveal that, even
with some optimization, given N allocated blocks,
another 0.5N blocks will be lost to fragmentation
This is fifty percent Rule.

One solution to the problem of external fragme
-tation is compaction. The goal is to shuffle
the memory contents so that all free memory
is together in one large block.

// Compaction is not always possible. If relocation is
static & is done at assembly/load time, compactio
can be done b/c the address manipulation is
not possible.

// It is possible only when relocation is dynamic
& done at execution time.

Solution to fragmentation Paging :-

→ solution to external fragmentation is to permit the logical address to be non contiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

Basic Method to Implement Paging :-

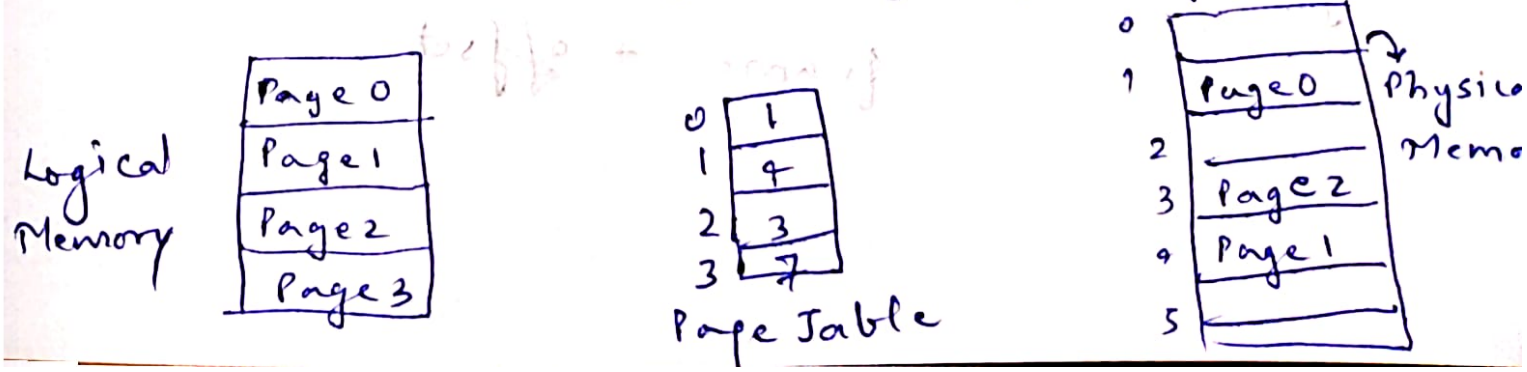
Physical Memory ⇒ divided into frames (the area left after OS is divided into loaded frames).

Logical Memory ⇒ divided into blocks of small size called pages.

When a process is to be executed, its page are loaded into any available memory frame from their source)

// Paging ⇒ Memory management scheme that permits the physical address space of a process to be non contiguous.

Hardware support for paging :-



Every address generated by CPU is divided into two parts ⇒ a page number (p) & page offset (d).

The page number is used as index into the page table. The page table contains base address of each page in physical memory. This base address is combined with page offset to get the physical memory that is sent to memory unit.

If the size of logical address is 2^m & a page size is 2^n addressing units, then the high order m-n bits of a logical address designate the page number, & n lower order bits designate page offset.

logical address / size of physical frame.

↓ in

mapped index page table entry gives physical frame.

Physical address = physical frame no * size of physical frame + offset.

When we use a paging scheme, we have no external fragmentation. Any free frame can be allocated to the process needing it. However we may have some internal fragmentation.

In worst case, a process would need n pages plus 1 byte. It would be allocated n+1 frames & internal fragmentation will occur.

If process size is independent of page size, we expect internal fragmentation to average one half page per process.

An important aspect of paging is the clear separation between the user's memory view & actual physical memory. The user program views memory as one single space, containing only this one program.