Unit-3 Linker & Loader

	Static Link Library	Dynamic Link Library
1]	The library code is copied into the program when it is compiled.	The library code is not copied, it is used at runtime
2]	Each program has its own copy in memory.	Multiple programs can share the same lib in memory.
3]	To update the library, you need to recompile the program	You can update the library without recompiling the program.
4]	The executable file becomes bigger	The executable file is smaller
5]	Performance is faster because calls are directly linked	Performance is slower due to runtime linking.
6]	Easier to distribute only the executable is needed	Need to distribute both the executable and the DLLs it depend on.
7]	No external dependency once compiled	Needs the DLL present at runtime, else error occurs.
8]	less portable, since large exe must be moved.	more portable, small exe but must carry DLLs
9]	file extension - .lib files in C/C++	file extension - .dll files in windows

• Need of DLL -

1] Saves memory -
- One DLL can be shared by many programs.

2] Makes updates easy -
- Only DLL file needs to be changed, not the whole program

3] Reduce file size -
- No need to copy same code into every program

4] Gives flexibility -
- DLL loads only when needed.

5] Reusability -
- Same DLL can be used by many applications

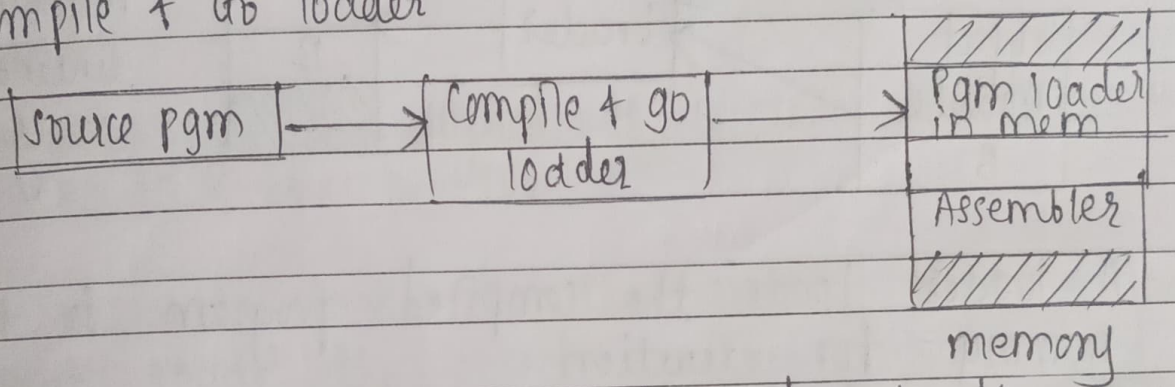
6] Supports modular programming -
- Program divided into small parts (DLLs).

Loader - It loads the compiled program into memory for execution
Loader functions - ① Allocation ② Linking ③ Relocation ④ Loading

- Types of loaders -

① Compile & Go loader
② General loader
③ Absolute loader
④ Relocating loader

① Compile & Go loader

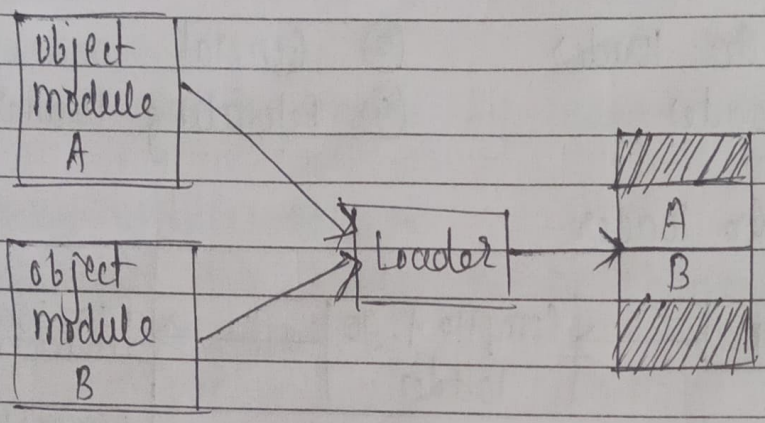


- source program is compiled and directly executed without producing object file
- Compiler itself acts as the loader
- The program is compiled & immediately executed
- No separate object file is created

Adv - simple & fast for small programs
No need to store object files

Disadv - Recompilation required for every run
waste of memory & time for large programs
Difficult for debugging & linking separately

② General loader



- A loader loads the compiled program into the memory for execution.
- The general loader scheme shows the overall working of any loader - how it loads relocates, links & executes a program
- It performs four main functions -

① Allocation - The loader decides where in main memory each part of the program will be placed.
It allocates space for all modules & data

② Linking - It connects different program modules

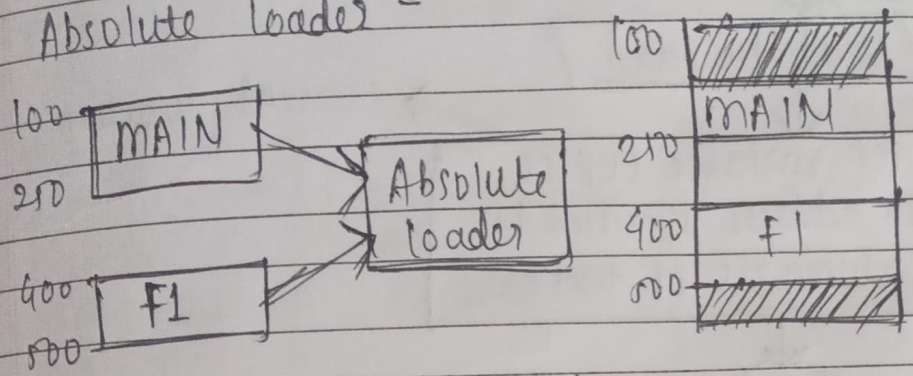
③ Relocation - Adjust all mem addresses in the program

④ Loading - finally, the loader places the relocated code & data into main mem & prepares it for execution.

Adv - Performs all functions automatically
Saves programmers time
Helps in efficient mem management

Disadv - Complex design
- Needs extra mem space
- Debugging is harder

③ Absolute loader :-



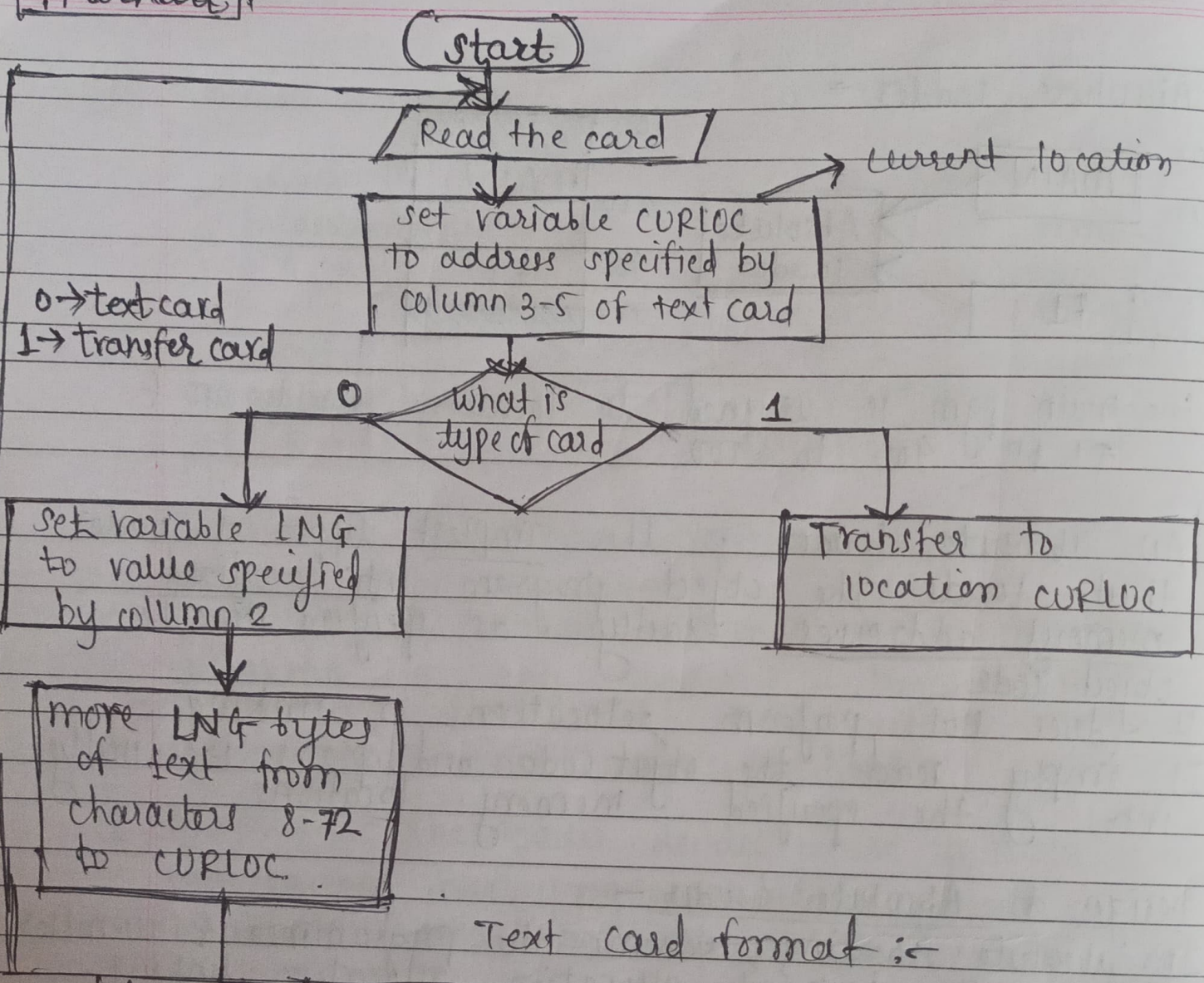
- The main pgm is assigned to locations 100 to 250 & F1 fn 400 to 500.

- An absolute loader is the simplest type of loader that loads the object program into a fixed memory addresses exactly as specified in the object code.
- It does not perform relocation or linking
- It simply read the object code and places it directly into the specified memory locations

• Design of Absolute loader -

- In absolute loading scheme the programmer & assembler perform the task of allocation, relocation, linking.
- Hence, task of loader is simply to read object pgm from assembler and to place the translated code at the absolute locations specified by the assembler
- Every object code consists of 2 types of information. This inform^n is present in 2 types of cards. Hence loader always read from 2 types of cards from translated object file. first type is text card & 2nd is tranfer card
- On text card, there are m/c instrn created by assembler along with absolute addresses. loader simply places these instrn in memory at specified absolute addresses
- On tranfer card, entry point of pgm is mentioned.

Flowchart :-



Text card format :-

Value	Column	Content
0	1	Type (0= for text card)
06	2	length of code (6 bytes)
200	3-5	Address . starting mem address to load data.
—	6-7	empty
141033281030	8-72	Instrn & data & to be loaded in memory
000001	73-80	sequence no.

- Transfer card format -

Column	Content	Description
1	Type	1 → Transfer card
2	count = 0	
3-5	Address	location where pgm execution begin
6-8072	Empty	Not used
73-80		Card sequence number

eg.

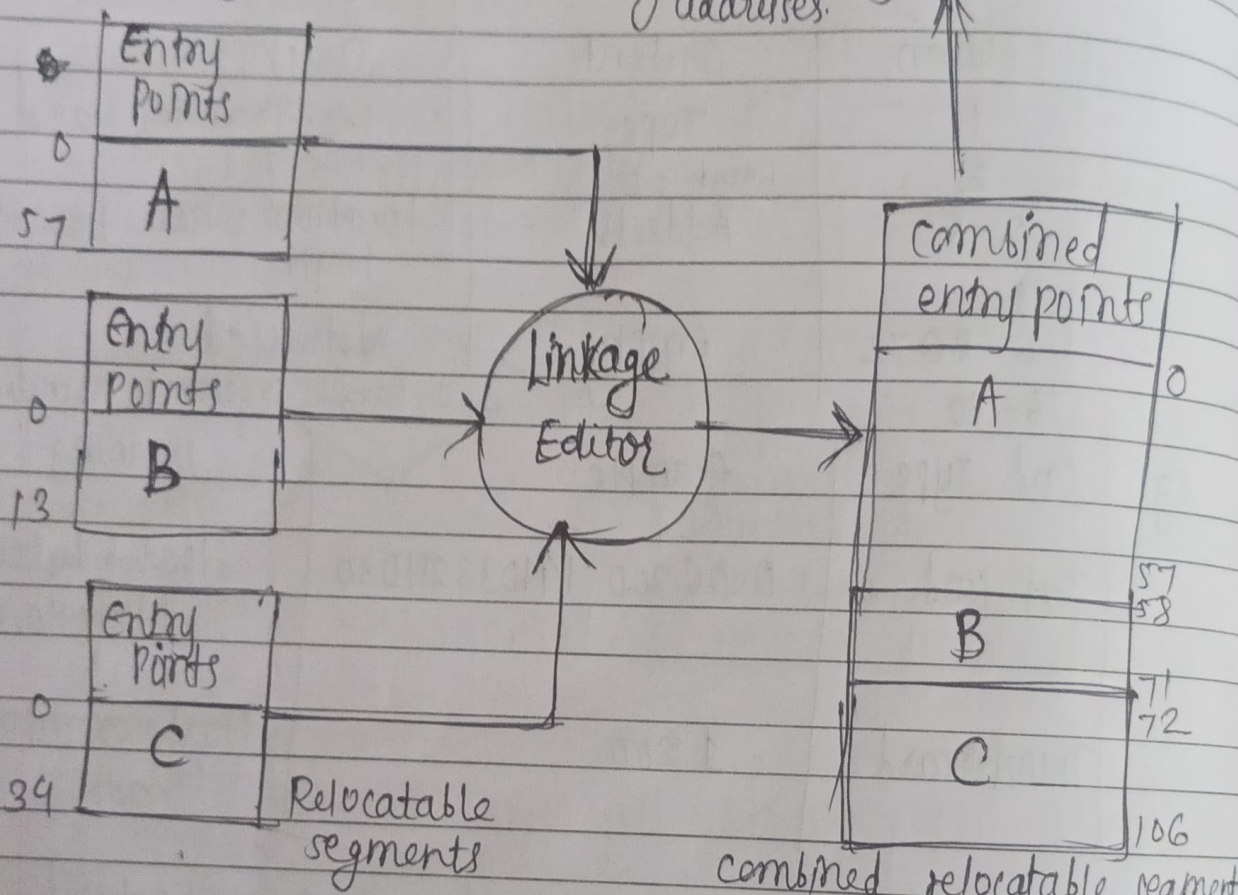
Card Type	Example	meaning
Text card	0 06 200 141033281030	load 6 bytes at address 200
Transfer card	1 200	start execution from 200

Adv. - simple design - easy to implement & understand
- fast loading no relocation or linking needed
- direct execution - program can start immediately after loading

Disadv - Not flexible - program must always load at the same address
- NO relocation or linking - can't combine multiple modules
- Diffucult to modify or debug - because addresses are fixed

④ Relocating loader

The combined segment can then be loaded anywhere in memory by relocating all internal addresses.



- A relocating loader is introduced to avoid reassembling all subroutines whenever a small change is made in one subroutine.
- It allows each program or subroutine to be loaded at any location in memory

Working - The assembler generates :-

① The obj program &
② Relocation information - which tells the loader which addresses need modification if program is loaded at a different location

- The relocating loader uses this information to adjust all address - dependent instructions

- A linkage editor combines multiple programs or subroutines (A, B, C, etc) into a single relocatable segment

- Binary symbolic loader (BSS) is an example of relocating loader. It allows many code & one data segment.
- Starting memory address is stored in the segment register & actual address is given by

Segment register + offset = Actual address

Adv - Avoids reassembling subroutines
- Program can be loaded anywhere in memory
- Supports modular programming

disadv - Loader design is more complex
- Takes extra time for execution

Q. Subroutine linkage

- Subroutine linkage is the method of transferring control from one part of a program (main pgm) to another (subroutine) & returning back after the execution.

How it works -
- When a subroutine is called, the return address is stored
- Control jumps to the subroutine
- After the subroutines finishes, control returns using that saved address

- Assemblers & linkers/ loaders use two special statements

1. ENTRY statement
- Used in called subroutine
- It declares a symbol that can be accessed by other modules.

2. EXTRN statement
- Used in calling program
- It declares a symbol defined in another module.

- Example -

module 1 (main program)

```
EXTRN SUB1
CALL  SUB1
```

module 2 (subroutine)

```
ENTRY SUB1
SUB1: ....
      RET
```

- The loader links CALL SUB1 from module 1 with SUB1 defined in module 2.
- So, both statements are used to connect subroutines across different modules during loading / linking.
- Allows code reusability & modular programming

- Overlays -

- Overlays are a memory management technique used when the program is larger than available main memory
- only the required part (overlay) of the program is loaded into memory at a time.

- Program is divided into smaller parts (overlays)
- loader replaces one overlay with another as needed during execution.

- main program → overlay 1 executes → replaced by overlay 2
- Helps to run large programs in limited memory.

eg. consider a program containing 5 subroutines A, B, C, D, E which require total 100kb memory
subprogram A calls B, D, E
B calls C, E
D call E
C & E do not call.

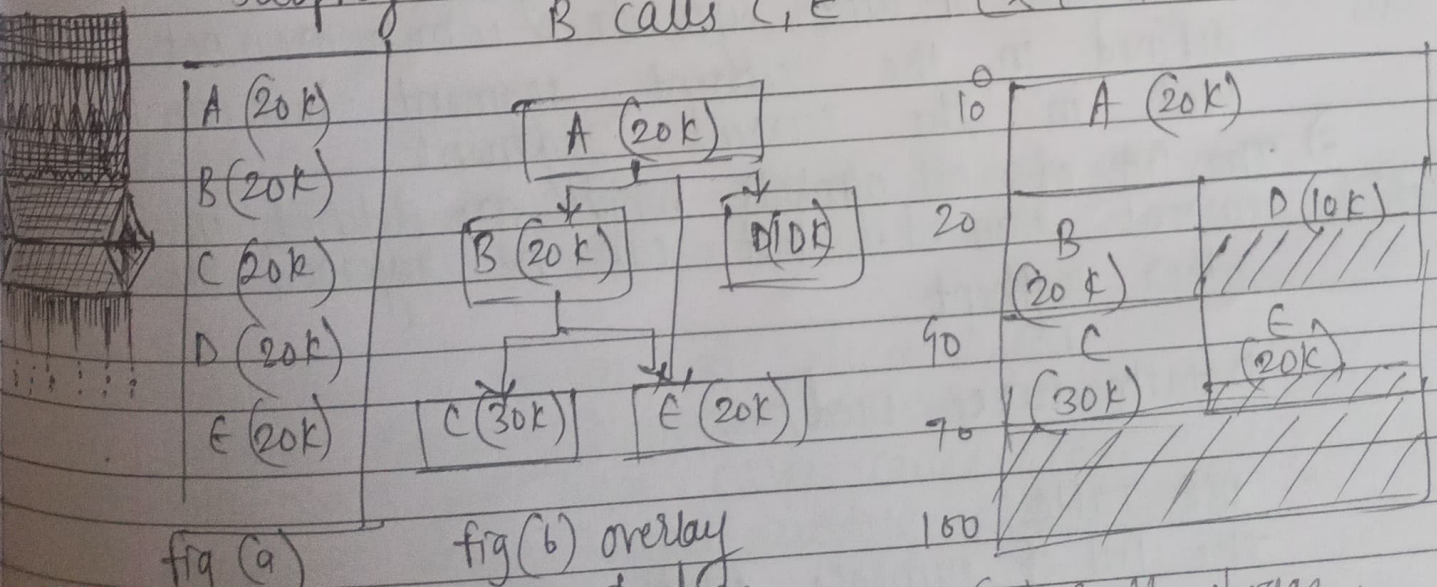


fig (a)

fig (b) overlay structure

(c) Possible storage assignment of each procedure

Q. Direct Linking Loader

- A direct linking loader is general relocating loader that is highly popular because it allows programmers to use multiple procedure & data segments & provides flexible intersegment referencing.
- It performs the critical functions of a loader -
  - Allocation - assigning memory space
  - Relocation - adjusting addresses
  - Linking - resolving symbolic references
  - Loading - physically placing the code into memory

- Design of DLL -

- The assembler should give following information to the loader :-

1) The length of object code segment
2) The list of all symbols which are not defined in the current segment but can be used in the current segment (EXTRN)
3) The list of all symbols which are defined in the current segment but can be referred by other segment (ENTRY)

- Datastructures used :-

- USE Table -

The list of symbols ~~which~~ which are not defined in the current but can be used in current segment are stored in a data structure

called USE table.
- The use table holds the information such as symbol name, address, address relativity.

- Definition table :-
- The list of symbols which are defined in the current segment but can be referred by the other segments are stored in a data structure called DEFINITION TABLE.
- The definition table holds the information such as symbol name, & address

- Phases of DLL Design -

- Pass 1 -
- Input - Object decks & Initial Program Load Address (IPLA)
- Assign addresses to each control section
- Build external symbol Dictionary (ESD) using ESD cards
- Record segment addresses, symbol names & lengths.

- Pass 2 -
- Perform relocation and linking using RLD cards
- load actual instructions from TEXT cards into memory

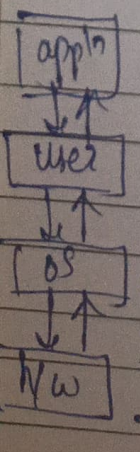
There are 4 sections of object Deck -
① External symbol Dictionary cards (ESD)
② Instruction and data cards called "TEXT"
③ Relocation and linkage directory cards (RLD)
④ END card (END).

Unit-5 Concurrency & Synchronization Control

- Need of Process Synchronization -
  - Process synchronization is needed when multiple processes access shared resources concurrently.
  - Without synchronization, processes may interfere with each other & produce incorrect or unpredictable results.

1. To avoid Race conditions
2. To maintain Data Consistency
3. To protect Critical sections
4. To coordinate process execution
5. To prevent deadlocks & starvation
6. To improve system efficiency
7. To support inter Process Comm^{n} (IPC)

- OS :- Operating system is a system software that acts as an interface betn the user & the computer hardware.
  - It manages h/w resources, control program execution & provides essential services to users & applications.



- Roles of OS -
  - manages CPU, memory, files, and I/O devices
  - Provides system security
  - Handles multitasking & multiprogramming
  - Enables user interaction.

Operating system services -

- The OS provides various services to make the system convenient & efficient. These services are -

① Program Execution :-
- OS loads program into memory, executes them, & handles their termination.
- It ensures correct & reliable program execution.

② I/O operations -
- Programs need to perform input & output
- OS provides I/O routines to handle these operations safely

③ File system management -
- OS manages files and directories on storage devices.
- It provides services like -
  file creation, file deletion, Reading / writing, Access control, Directory management.

④ Memory management -
- The OS manages primary memory (RAM).
- It allocates & deallocates mem to processes & ensures efficient use of mem using techniques like Paging, segmentation, Swapping.

⑤ Process management -
- OS manages all processes (running programs)
- It performs - Process creation / deletion, CPU scheduling, synchronization, deadlock handling, IPC.

⑥ Device management -
- OS controls & communicates with h/w devices using device drivers.
- It handles - device allocation, Buffering, Caching, Scheduling of I/O devices.

⑦ Protection & Security -
- OS ensures safety of files, mem & resources
- It provides - Authentication, Authorization, protection against malware, unauthorized acess & errors.

⑧ Error detection & Handling -
- OS constantly monitors system activities
- It detects & errors such as -
Hardware failures, mem errors, file system issues
- Then it takes corrective actions to ensure stable system operation.

• CPU scheduling -
- CPU scheduling decides which process gets the CPU and for how long.
- there are two types of scheduling.

① Pre-emptive scheduling -
- Preemptive scheduling means the OS can stop a running process & give the CPU to another process (interrupt)

Key features - OS can take away the CPU anytime
- Gives better response time

Non-Preemptive	Preemptive
FCFS	SJF
SJF	Priority
Priority	Round Robin

- Suitable for time-sharing & real time systems.
- Higher priority processes can immediately execute.

. Adv - Good for interactive systems (windows, linux)
- No long waiting, urgent tasks can run quickly
- more flexible and fair.

. Disadv - more overhead due to context switching
more complex to implement

: Examples of Pre-emptive algorithms -

① Round Robin (RR) ② Shortest Remaining Time first (SRTF)
③ Pre-emptive Priority Scheduling

. If Process P1 is running & a new process P2 arrives with higher priority → OS stops P1 & gives CPU to P2.

② Non-Preemptive Scheduling -

- In non-preemptive scheduling, once a process gets the CPU it cannot be stopped until it finishes or voluntarily releases the CPU.

Key features - Process run till completion or till it waits for I/O
- No interruption by OS.
- simpler & easier to implement

Adv - No overhead of context switching.
- very simple & predictable.

Disadv - Not suitable for interactive systems
- long waiting time if long process is running
- may cause starvation for short jobs.

Eg. of non-preemptive algorithms -
1. First come first serve (FCFS)
2. Shortest Job first (SJF)
3. Non pree-emptive Priority Scheduling

- If P1 is running even if P2 with higher priority arrives, P2 must wait until P1 finishes

1. Semaphore -
- A semaphore is a synchronization tool used to control access to a shared resource.
- It uses a integer variable and two operations wait() & signal().
- Types of semaphore -
  1) Binary semaphore :- (value = 0 or 1)
  2) Counting semaphore - value can be more than 1 (for multiple resources)
- ~~Characteristics~~ operations -
  - wait() - decreases value; if value < 0, process waits
  - signal() - increases value; if value ≤ 0, wakes a waiting process

working - (1) Initialization - A semaphore is initialized with a non-negative integer value, representing the no. of available resources.

(2) wait operation (P) - When a process wants to access a resource, it calls wait () opn
- If semaphore value is greater than zero the process can access the resource, & the semaphore value is decremented by one
- If semaphore value is zero or less, the process must wait until resource becomes available.

(3) signal opn (V) - When a process finishes using the resource, it calls the signal () opn
- This opn increments semaphore's value, indicating that resource is now available for other waiting processes.

- semaphore is used because to -
1) Prevent race condition
2) Ensure mutual exclusion
3) Process synchronization
4) Avoiding deadlocks

(2) monitor
- A monitor is a high-level synchronization construct that contains -
· Shared data, · Procedures to acess the data, A lock
- only one process can be inside the monitor at a time.

- It is easier & safer than semaphores
- It uses conditions variables with operations:
  - wait()
  - signal()
- Java, C++, VB are some of langs that allow use of monitors
- Synchronization methods like wait() & notify() construct available in Java lang.

Characteristics of monitor in OS -

1. We can only run one program at a time inside the monitor
2. A pgm cannot access monitor's internal variable if it is running outside monitor
3. monitor were created to make synchronization problems less complicated
4. Monitors provide a high level of synchronization bet^n processes.

- Mutex - (mutual exclusion lock)
- A mutex is a binary lock used to give exclusive access to one thread or process at a time
- mutex ensures that only one thread can access to a critical section or data by using opⁿs like lock & unlock
- lock() → if free, the process gets the lock
  unlock() → releases lock
  only one process can hold the mutex
- mutex is a special binary semaphore that synchronizes the access to shared resource like memory or I/O.

- only the thread that locks the mutex can unlock it

Adv - No race condition arises.
- Data remains consistent & it helps in maintaining integrity
- It is a simple locking mechanism that into a C.S & is released while leaving C.S.

disadv - sometimes leads to starvation
- when previous thread leaves C.S then only other processes can entire into it, there is no other mechanism to lock or unlock C.S.
- busy waiting, leads to wastage of CPU cycle

- Deadlock -
- Deadlock is a situation where two or more processes are waiting for each other forever, & none of them can continue. The system is stuck forever.
- Process P1 has Resource R1 & needs R2
  Process P2 has Resource R2 & needs R1
  Both wait for each other ⇒ deadlock

- Conditions (All must occur)
- Also called Coffman Conditions.
- for deadlock to happen/occur, the foll^n 4 conditions must hold simultaneously.

① Mutual Exclusion
- only one process at a time can use the resource.
- If another process requests the resource, the requesting process must be delayed until the resource has been

released.

eg. Printer - only one process can use it.

② Hold & wait -

- A process is holding at least one resource & is waiting to acquire more additional resources that are currently held by other processes.

eg. P1 holds R1 & waits for R2.

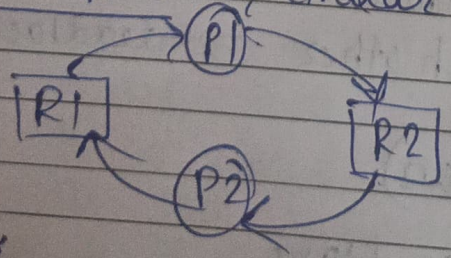
③ No Preemption -

- A resource cannot be forcibly taken; a process must release it voluntarily, after that process has completed

eg. OS cannot snatch printer from P1 until it finishes printing

④ Circular wait -

- A set of processes are waiting for each other in circular form. / circular chain.



- If all 4 conditions are true at the same time, deadlock definitely occurs.
- Breaking any one of them prevents deadlock

Deadlock prevention -

- Deadlock prevention means ensuring that at least one of the four conditions never occurs.

① Prevent Mutual exclusion -

- Make resource sharable if possible
- But many resources cannot be shared (printer, tape drive, file writing)
- This method is rarely practical

② Prevent Hold & wait -

- The idea is -

A process must request all required resources at once, before it starts!

OR

A process holding some resources must release all before requesting more.

eg. A process needs: printer & file & tape drive → it must request all three before starting.

Benefit - No "holding some & waiting for others" → breaks hold & wait.

Disadv - Resource remain unused for long.
Processes may starve if they cannot get everything at once.

③ Preventing No Preemption -

Idea is - If a process holding some resources cannot get a new resource, it must release all currently held resources.

eg. P1 holds memory block M1
It requests M2 → not available
→ OS forces P1 to release M1 & wait again

These prevent deadlock because resources are not permanently held

disadv- Released resources may cause loss of work.
frequent preemptions reduce efficiency

(4) Preventing Circular wait.
idea is - Impose a fixed ordering of resources
- every process must request resources in increasing order only.

eg- Resource order - R1 < R2 < R3 < R4

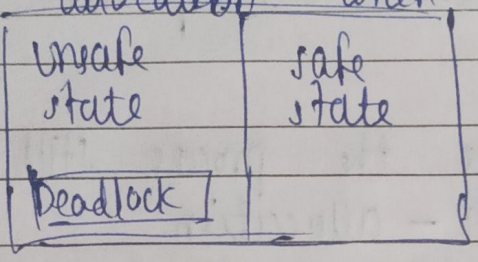
- A process needs R2 & R4 → must request R2 → then R4
Cannot request in reverse order

- No circular chain can form when all processes follow the same order.

Adv - Simple & effective . Prevents circular wait completely
disadv- Processes sometimes req resources earlier than needed

- Deadlock Avoidance.
- Deadlock avoidance does not prevent deadlock always
- It ensures the system never enters an unsafe state.

- The most famous technique is Banker's Algorithm.
- The system must be able to decide whether granting a resource is safe or not & only make allocation when it is safe.



- Each process must declare its maximum resource requirement in advance.
- The OS allocates a resource only if the resulting state is safe.
- A safe state means there exists a safe sequence of process execution where all processes can finish.

- **Banker's Algorithm** -
- It is a deadlock avoidance algorithm used in OS.
- It decides whether a resource request by a process can be safely granted or not.
- It is called banker's algorithm because it works like a bank giving loans only if the bank will still be safe after granting the loan.

Why Banker's algo?
- Deadlock avoidance → the OS checks future possibility of deadlock before allocating resources.

If allocation keeps the system in safe state → req is granted
If it makes system unsafe → req is denied temporarily

① maximum need -
- The maximum no. of resources a process may ever req

② Allocation -
- Resources already allocated to a process.

③ Need -
- Remaining resources the process still requires.
Need = maximum - allocation

④ Available -
- Resources available in the system right now.

⑤ Safe state -
- A system is in a safe state if there exists a sequence in which all processes can finish without deadlock.

· Steps :-

① Check need ≤ Available. -
- If a process requests some resources -
Check if
Request [i] ≤ Need [i]
and
Request [i] ≤ Available
if not → deny request.

② Pretend to allocate -
Temporarily allocate resources & update tables -
Available = Available - Request
Allocation [i] = Allocation [i] + Request
Need [i] = Need [i] - Request

Available = Total - Allocated.

③ Check Safety
- Check whether the system is still in a safe state
- try to find a sequence where each process can finish one by one
- If a safe sequence exists → final allocation is allowed.
If not → undo temporary allocation.

eg. system has - 1 type of resource (A) & total 10 units
~~Total resources of type A = 10~~
Three processes - P1, P2, P3.
Processes and their data:

Process	max	Alloc	Need
P1	7	3	4
P2	5	2	3
P3	3	2	1

Available = 10 - (3+2+2) = 3
so, we have 3 resources free right now.

- Safe sequence check -

① P3 needs 1 ≤ Available 3 → can execute
Available = 3+2 = 5

② P2 needs 3 ≤ Available 5 → can execute
Available = 5+2 = 7

③ P1 needs 4 ≤ Available 7 → can execute
Available = 7+3 = 10

∴ Safe sequence : P3 → P2 → P1
- System is SAFE
- No deadlock.

- This algo is safe because it never grants a request that may lead to deadlock
- It checks future possibility before giving resources.

Adv- Avoids deadlock completely.
ensures system remains in safe state

disadv - Complex to implement
- Requires knowing max needs in advance
- Works only for fixed number of resources

• Hardware Approach for Mutual Exclusion :-

mutual exclusion - Only one process can access a critical section at a time
- H/w provides a special instructions that help achieve mutual exclusion at the CPU level.
- These instructions are atomic (cannot be interrupted while executing).

• Hardware Instructions used for Mutual Exclusion -

1. Disable Interrupts
2. Test & set lock (TSL / TS / TST)
3. Swap Instruction.

① Disable Interrupts -

Idea - The process turns of all interrupts while entering the critical section.

When interrupts are off -
- CPU cannot switch to another process
- so only the current process runs.
- mutual exclusion achieved

Steps - 1. disable interrupts
2. enter critical section
3. execute
4. enable interrupts again

dis-adv - Dangerous : If a process forgets to enable interrupts the system will stop.
- only works on single-CPU systems
- Processes get too much power (can block CPU entirely).

② Test & set lock (TSL Instruction)

- This is the most important h/w approach

Idea - TSL checks and sets a lock in one atomic operation.

- A lock variable (0 or 1) is used -
  - 0 - free
  - 1 → busy

- TSL instrn (atomic) :

TSL R, LOCK - means.
load Lock into register R
set Lock = 1
Both happen together without interruption

Process -

```
do {
    TSL R, lock
} while (R==1) ; // wait if lock was already taken
// critical section
lock = 0 ; // release
```

- If lock was 0 → process gets access
  If lock was 1 → someone is already inside → keep waiting
- It works because, TSL is atomic → no two processes can execute it at the same time.

(3) Swap Instruction -

Idea - Swap values of two variables ~~automatically~~ atomically.

eg. swap (lock, key)

- lock = 0 means free
  key = 1 means process wants entry
- Process keeps swapping until lock becomes 0.
- Process keeps asking:
  "Is the lock free? Yes? Then take it".

Adv of h/w approach - Very fast
- No complex s/w logic needed
- Atomic operations guarantee mutual exclusion
- works on multiprocessor systems
- Solves race conditions effectively

disadv - Busy waiting
- starvation possible
- disabling interrupts is dangerous
- cannot ensure bounded waiting
- some instrn require special hardware support

- Producer Consumer Problem -

- The Producer-Consumer problem is also called the Bounded Buffer Problem.

Concept - we have a buffer (fixed size storage)
Producer adds items into the buffer
Consumer removes items from the buffer

The problem -
Producer must not add items if the buffer is full
Consumer must not remove items if buffer is empty
Both must not use the buffer at the same time → mutual exclusion

Semaphores used - mutex = 1 → ensures one process at a time in buffer
empty = N → counts empty slots in the buffer
full = 0 → counts filled slots.

- Solution - Using Semaphores

Producer code

```
do {
    produce_item();
    wait(empty);    // wait if buffer full
    wait(mutex);    // enter critical section

    insert_item();  // add to buffer
    signal(mutex);  // exit critical section
    signal(full);   // increase filled count
} while(true);
```

Consumer code -

```
do {
    wait(full);
    wait(mutex);

    remove_item();
    signal(mutex);
    signal(empty);

    consume_item();
} while(true);
```

- Producer waits until empty > 0, consumer waits until full > 0
- mutex makes sure only one uses the buffer at a time

Imp bcoz - Demonstrates synchronization betn two processes sharing a common resources.

• Dining Philosopher Problem -

- The Dining Philosopher Problem was given by Dijkstra
- It explains the issue of deadlock, starvation, and synchronization when multiple processes share limited resources.

Problem statement - There are 5 philosophers sitting around a round table
- Betn each pair of philosophers, there is 1 fork
- To eat, each philosopher needs 2 forks:
  one from the left
  one from the right
- Philosophers do two actions -
  Think → eat → Think → eat

Problem - If all philosophers pick their left fork at the same time, no one can pick the right fork.
- Everyone waits forever → deadlock

Soln using Semaphores -

We treat each fork as a semaphore.

`fork[i] = 1 for all i`

1 → fork is free 0 → fork is being used

Initialization semphore fork[5] = {1, 1, 1, 1, 1}
semaphore room = 4

Philosopher Process -

```
do {
    think ();
    wait (room);          // allow only 4 philosopher
    wait (fork[i]);       // pick left f
    wait (fork[(i+1) % 5]);   // pick right f

    eat ();

    signal (fork[(i+1) % 5]);
    signal (fork[i]);

    signal (room);
} while (true);
```

- avoids deadlock
- avoids circular wait