Linux & Embedded OS Notes – Lalit Deokar

-:Operating Systems:-

Overview:-

- An operating system is acts like a intermidiate between the user of the computer and the computer hardware
- purpose of a operating system is provide an environment in which user can execute a program in convient and effiecent manner.
- O.S is a program that manages the computer H/W
- The design of the new operating system is a major task it is impotant that the goal of the system will be predifined.

Defination & Role of O.S:-

- Definition:- O.S that manages a computer H/W resources such as cpu, memory and storage devices, ensure that this resource allocated effectively to various program and processes.
- Role:- The OS acts as an intermidiate between the user and Hardware.

Variability in Operating Systemes:-

- Mainframe O.S:-
  - Optimized for hardware utilization, enabling effecint processing of large volumes of transaction and data.
- Used in critical application, such of banking and large-scale processing.
- PC OS:-
- supports a wide variety of applications from productivity tool to complex games, it includes ~~OS~~ OS like Windows, Linux and macOs.

Embedded operating system:

- Embedded operating system are designed to operate on embedded system.
- These systems are often used for application where limited Hardware resource (like memory & processing power. requires a O.S.
- Eos found in a device that perform specifin task in a automotive control systems, medical device home appliance, industrial machine, and IOT devices.

Why we need EOS:-

- Essential because they provide lightweight, effienent and reliable platform for running a software on a devices with specific critical function.
- Eos is optimissed for resource contrained enviroment where reliability realtime responselvess are minimul power consumtion are essential.
- Resource optimisation:- Eos devices are having limited memory processing power and battery life An eos is designed to use these resources effiently allowing the device to perform well without extra load.
- Real-time Operation:- Embeded systems are needed to responced instantly.

For example:- Air bag systems in car needed to deploy within a second (Rtos ensures that the critical deadline are meat.)

what makes eos differe from GPOS.

1. Purpose & Functionality.

- Eos :- designed for specific task within device
- GPOS :- Designed for versability, allow user to run wide range of application.

2. Resource Effiency :-

- Eos : Uses minimum resource (memory, processing and storage). are limited H/W. EOs is highly optimised are often stripped down to essential functions.
- GPOS :- Has more resources available and support complex application and multitasking which require more memory and power.

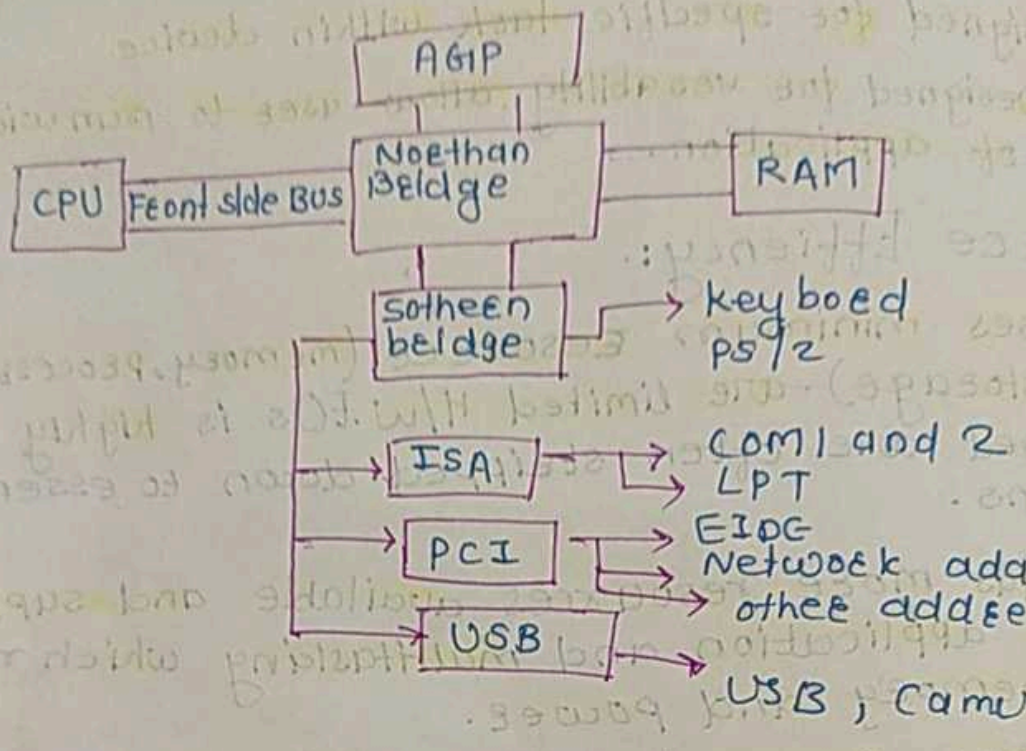
3. Real-time Capability :-

- Eos : are handle time sensetive task with strict deadline
- GPOS : They are generally less focused on a gurented responsed time, as delay are more acceptable in most tasks.

4. Reliable and stabality

- Eos : Most be highly reliable and stable without human iteraction runs critical task.
- GPOS :- Relibily inportant, But less focused on a stability over long-time operation device must needed tobe restart or update.

Electronics : PC Architecture:



- CPU:- Executing instruction and perform calculation. Interprete and process data from ~~[illegible]~~ appl. and the O.S.
- Front Side Bus
  - It is the primary connection between CPU and RAM and other components of motherboard
  - It transfer data between Northan Bridge and CPU which communicate with ram and other peripherials
- Northenbridge (Memory control Hub):-
- The northanbrigd chip is manages high-speed communication CPU, RAM and AGP
- Its act like a intermediatory helping CPU to access the memory and graphics card.

RAM:- short term memory used by CPU to store data instruction needed for current process.

AGP (Asselarated Graphics port):
- AGP is older bridge interface which designed for connect graphics card to directly Northan bridge.
- Allow faster data transfer rate between component for improve the rendering.

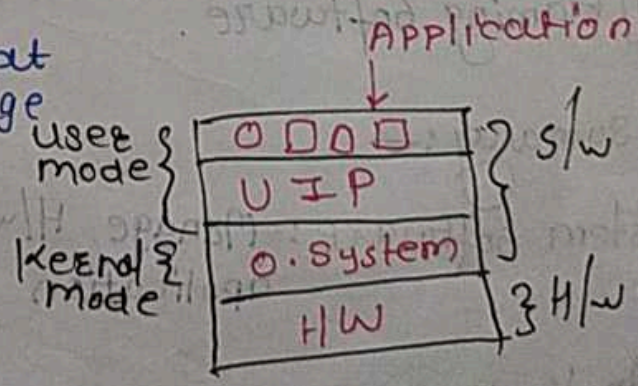
Southern Bridge:
- that manages slower peripheral device connections and serves as a bridge to input/output device like storage, Network adapter, and USB ports.

—xox—

Impotant points:-
- All computer components and all devicesed need to be managed so its done by a layer of s/w running that called as O.S (Operating System).
- Computer sytem = hardware + software
- Software = Application s/w + system s/w.

Computing :- System Layer and O.S.

The O.S complex program that
- Operate directly on H/w manage system resource
- Its act a like Intermeditory
- Enable User program to interact with H/w through its Interface (O.S).



System software and Application Software:

·Sytem sofware :-

- Sytem software serve as a foundation for running a ~~software~~ computer and manages H/w so that application software can function.

Examples:- i] OS:- provides interface bet user, appli-software and H/w.

ii] Device Driver:- These are spellised program allows the O.S to communicate with H/W components.

iii] Utility s/w:- That mange, maintain and optimised performance of computer

·Application Software:-

- It consists of Program designed to perform a specific tasks for the user.
- It depends on sytem software to function but provide functionality that user needed.

Examples:-

i/ Productivity Software:- MS office, G-Docs, VLC media.

ii] Web browser :- Browsing Internet, chrome, firfox

iii/ Gaming Software.

In Sumary :—

i. System software :- Manage H/W and create enviroment for application software to run.

ii. Application Software :- Enable users to perform specific task on top of the operating enviroment that create by System software.

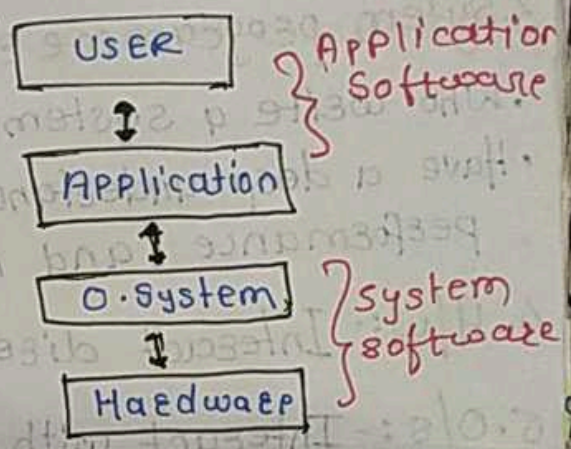
Computing : System Access :

In Computing :-

- The system access and H/W/S/W integeation eelay on a Layeeed Access.
- These steuctuee allows each layee to function indepedly theought define inteeface offeeing flenibily & stability and security in computing enviroment.

Layeeed Access :-

- Isolation layee ensuee that the change in one layee don't nesseeaey impact othee layee.



1. Scenaeios of changing H/W.

i) No change on Usee and Application layee and O.S.

ONy new H/W eequees compatible deivee to communicate with O.S.

2. Scenaeio of Installing New Software.

i) H/W and OS geneeally nemain unaffectable.

The new software must be compatible with ~~software~~ operating system.

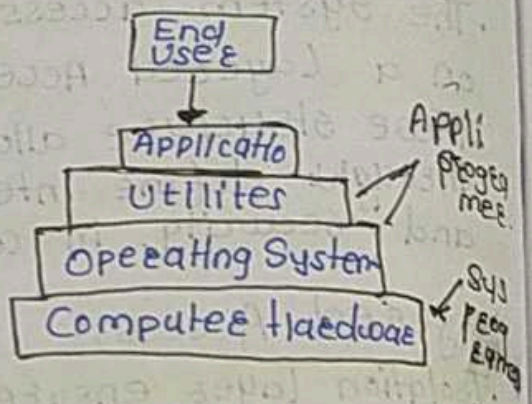
3. Changing oe upgeading O.S.

i. The O.S upgeade eequees compatible Appli and H/W.

ii. Deivee compability.

Computer Interaction: Human Roles:

Level of Human Interaction with computers



1. End User
- Only interact with computer and accomplished the tasks on application.

2. Application programmer.
- Who writes a program & application.
- They involves design, test, coding maintaining s/w

3. Sytem programmer.
- Who write a system level software such as OS and EDD
- Have a deep understanding H/W and system software work on performance and manage system resource.

4. H/W:- Interact directly with O.S and Execut cmd and process

5. O/S:- Interact with managing memory, controling peripheral provide security and provide communication

6. Utilites:- Spellised program to help manage, maintain & control.
Ex:- Include:- File management tool. disk cleanup program antivirus s/w, backup utilites.

The main Summary:-

In this structure that end user typically interact with high level s/w (application), while system programmer and application programmer, are part of that layer that support those interactions, ultimedly driven by Computer H/W.

O.S: Layeeed System abstraction

The concept of layeeed ~~application~~ abstraction of an O.S in relationspit with Haedwaee | Applico Program

In this

1. Haedwaee :- They "ugly" means they aee complex and haedee interface
2. O.S :- It interaction with H/W.

Beautiful Interface:- In this side we check the interactive and easy part

Ugly Interface:- In this side we check complex and hard part.

Lab-topic

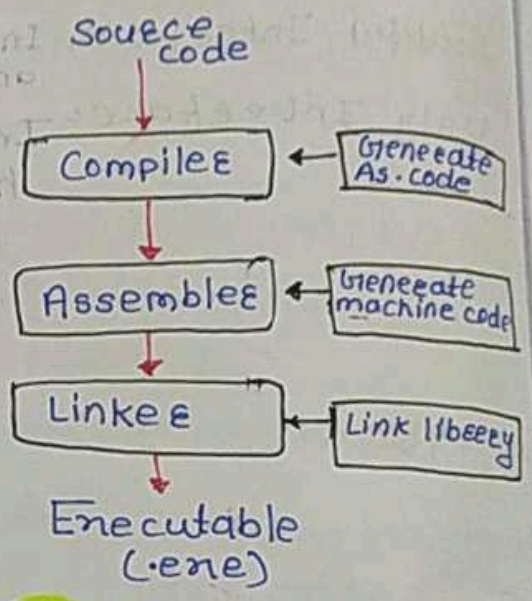
Static and Dynamic Linking... Process.....

Introduction:-

In this we learn the static and dynamic linking processes to generates the final executable from assembly code

first we see the generating Executable process.

- To execute any program we convert source code into machine code.
- We do this compiling the code to Intermidiate level and then converting assembly code
- After that we link the assembly code with other libraries it uses.



- Linking is the process combining external programes with our program to execute them successfully.

- After linking we get our final executable file.

Linker (Linking...):-

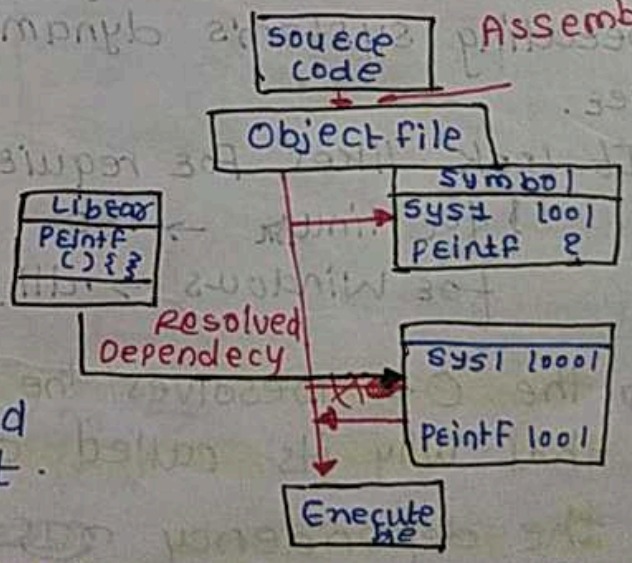
- The assembler translates our compiled code to a machine code and store the result in object files.
- object files are:- its a binary representation of our program.
- The assembler gives the memory location to each object and instruction
- Some of these memory location are virtual. They offset relate the base address of first instruction.

- Thats the case with refrences to external library
- The linker resolve them when combining the object files with libraries.

We use two machanisms for linking:
- Static Linking
- Dynamic Linking.

- Static Linking :-
- Lets look first dependencies so this are external codes like libraries or modules that a program relies on to function correctly.
- By using dependences, a program avoid duplicating fuctionallity.
- In static linking system linker copies dependencies into final executable at a time of linking external library the linker finds all dependencies are defined in the library.
- And replace them with corresponding functions from libray to resolve dependence in our code.
- After linker generate executable file that we execute on machine.

For Example:-
- Function call external fibray
- The assembler generate the ojt file with all native symbol resolved to their me. add.
- The external reference peint() can not be resolved
- The linker loads the libary and find defination of peintf init.



- Then its map the memory peintf in mm location and resolved dependency.
- So static linked file contain our program code as well as all libraries

Dynamic Linking:-

- Dynamic linking is process in which dependencies like libraries are not embedded directly into a final executable during linking stage.

- The linker only include references to the external library in executable file.
- When program runs the O.S loads when these external libraries are needed.

How Dynamic Linking works:-

1. Reference to share libraries:
   - During compilation and linking the linker does not copies the actual code of libraies into a executable instended of it includes references to the required external libraries often in the form of paths.

2. Load at Runtime:- After executing program the operating system's dynamic linker or loader takes over.
   - It look likes for required libraries - {shared libraies
     {for linux → .so
     for Windows → .dll} and loads theme into a memory.

So the O.S resolves the function, variable & symbol. that why its called dynaminc linking libraries the dependency ~~[illegible]~~ are linked at a runtime.

Onces all required libreys are loded all dependency are resolved the program manually.

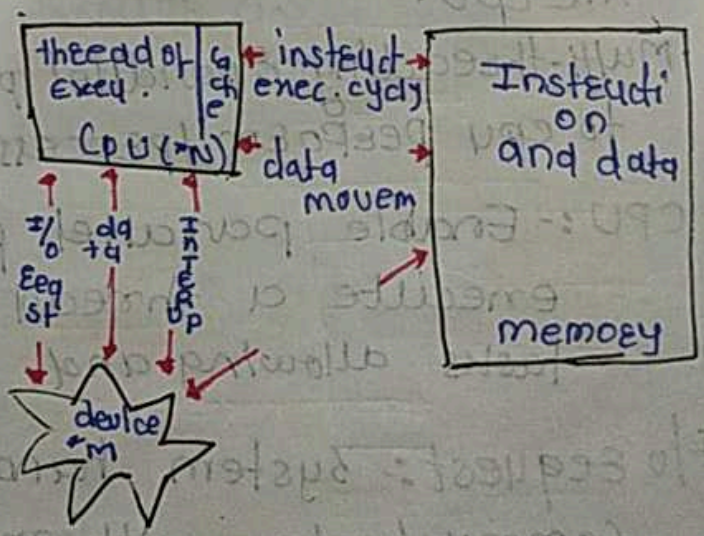
-: Computing System Data Movement :-

what is computing :-

- Computing refers to use of computers and other devices to process, stores and retrive data or perform a complex calculations
- It perform wide range of activites including programming, data analysis, and developing an algorithms as well as operating and managing software and hardware systems.
- At a core level computing allows user to automate tasks, solve problems, and handle vast a vast amount of information so far faster and more accurate than manual methods.

System General Workflow :-

- General flow of computing all system are similar at same part it differs.
- Floats of data movent can happens.



< On back page. >

1. Instruction and Execution Cycle.

- At the heart of computing is Instruction cycle Execution. also called fetch decode & Execute cycle.
- Instruction can involves arithmatic & logical operations, memory access. or data movement.

2. Data Movement:-

- Data movement is essential for computation all the component must be share data.
- HPC must be have optimize data flow to reduce bottlenecks (means that does not have required componet then its get overload on system) then system get slow.

cach:- helps to minimize delays by keeping high demanded data close to the processor, reducing access time for repeted instruction.

Thread of Execution:-

- A single sequence of instruction being execute by the cpu.
- Multi-threading & parallel processing allows/enable to cpu perform/process multiple tasks simultan.
- CPU:- Enable parallel processing. each core execute a thread with O.S managing tasks allowing and load balance

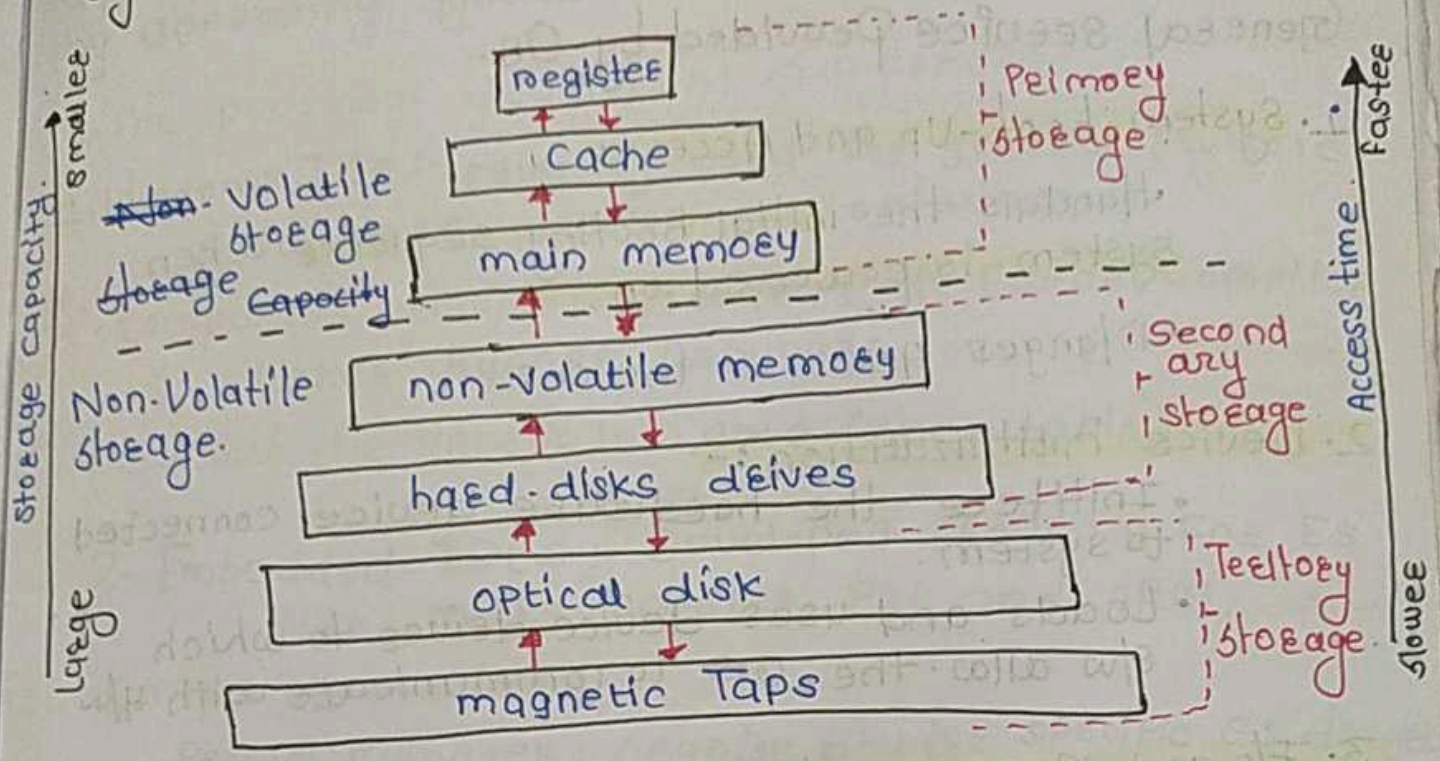
I/O request:- System handles I/O requests to communicates with external devices

Interrupts:- signals that pause the CPU's current work or handle more urgent task.

eg. I/O request, compilation signal, and error notification

Storage capacity → smaller

System : Storage and Hierarchy....



- Primary storage:-
  - These devices are closest to CPU.
  - Volatile (data is lost when power off).
  - Limited storage capacity.
- Secondary memory.
  - These devices accessed through O.S.
  - Non-Volatiles (data retain without Power).
  - storage larger than primary
- Tertiary storage.
  - Accessed through seperate system
  - Non-Volatile
  - storge capacity large
- As we go upward in Hierarchy the cost and Access time is increased. Access time (speed).
- And also the storage also get smaller from tertiary to primary storages.

-: Services :-

General service provided by Os.

1. System boot-Up and Access :-
   - Handals the initial booting sequence when system is powered on.
   - Manges Access to the O.S

2. Device initilization :-
   - Inititeze the hardware device connected to system.
   - Loads and uses device drives to which s/w allow the O.S to communicate with H/W

3. I/o and Management.
   - It manges i/p and o/p operation between various devices.
   - Manage disk, Directory and file system

4. Program Execution
   - Loads executable program in memory. allow nesessary resources and being execute.
   - Monitor program status, manages process life cycle and handles exit or exception events.

5. System Management.
   - Allow resources like CPU, memory, storage to different tasks and users.
   - Allow only authorised user to log in and use system resources.

Specialized services and environment provided by operating system:-

1. For program developent O.S provide.

1. Windows Ide :- primary used for general Purpose software development on windows.

- Componets provide:- Vscode, Code Editor, Compiler Linker, Runner, Debugger.
- Output file generate:- .exe (executable).

2. Embedded IDEs :- Developed program for ES like micro-controller PIC and ARM.

Componets provide:-

- Project Manager :- Orgnise files for specific ES target
- Code editor
- * Cross compilation:* Compile code for es plateform as opposed to host system.

→ Meaning

The host system are like development everiment running where code is written and compiles for example Windows, Linum PC. as hosy stem.

The target device like where the actually the program can run like a arm based controllers.

for example:- GCC for ARM.

we write code in HLLanguage (c/c++) on Linum and use arm-non-eabi-gcc for cross-compiler to generate executable files that will run arm device.

Output file that produced is tailored for specified ES.

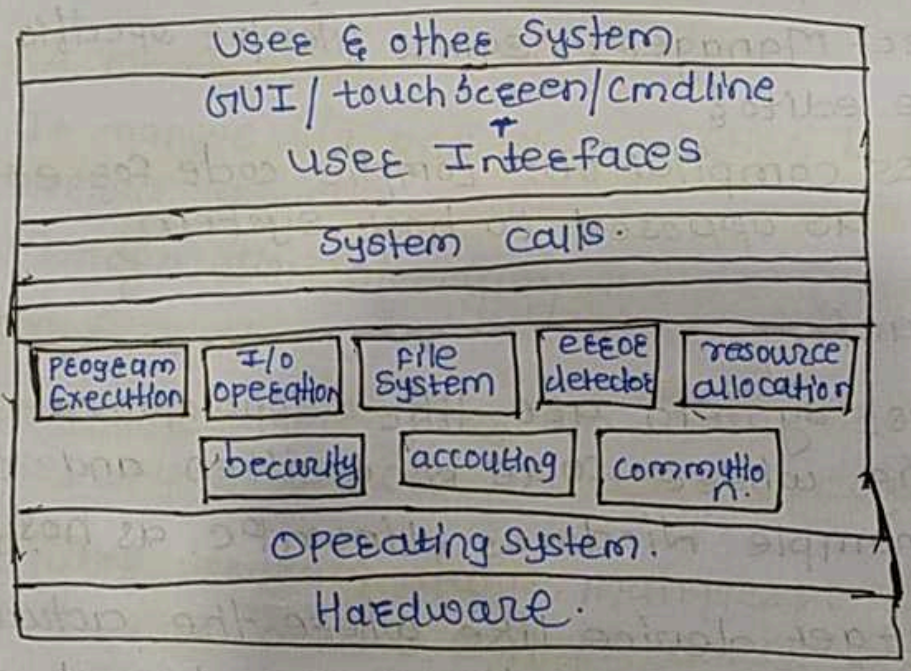
3. Kernal IDE... (OS programming)

Purpose:- Designed for low level programming of Operating system kernals.

Componets:-

- Kernal Development Tools:- GNU Toolchain.
- Kernal Debugger:- Tool like ~~KD~~ KGDB allowing debugging of kernal code.

OS services ~~illu~~ illustrate.



i] User and other system program

- This is the top layer, representing user application and other system program that the user interacts with.

2. User Interfaces:-

- Includes varies interaction or interfaces for user interaction.
- Example:- i] GUI (Graphical User Interface)
  ii] Touch screen
  iii] Command Line.

3. System Calls:-

This layer serve as the interface between user application and operating system's core services like a system Hardware.

4. Services:-

Represent O.S Functionalites that manages different aspects of system operation.

- Example:- i] Program Execution
  ii] I/O operations.
  iii] File systems
  iv] Communication.
  v] Resource Allocation
  vi] Accounting
  vii] Protecting and Security.

5. Operating Systems.

The main layer managing all services and system calls interacting directly with Hardware.

6. Hardware.

The physical components of the system that OS manages.

The layered architechture helps organizes sytem operations and interaction between user and applications and the Hardware.

what is difference between kernal programming and OS program:-

1. Kernal programing & o.s programing are closely related but they refere to different aspect of operating system development.

1. kernal programming:-

Focus:- Development and modification of kernal which is a core part of operating system

- kernal manages system resource and hardware and provide fundametal service to o.s and user application.

Role:-

- Process Management :- Schedule and fnecting processes.
- Manage Memory :- Allocating and managing Ram(Memory).
- Device Management: Interfacing with H/w device.
- Providing System calls:- Interface ~~with~~ for application request service from the o.s.

- A kernal programing is part of o.s programming.

O.S programming:- Involves developed all layers of operating system, from kernal to user interaction

Key difference :-

1. scope

- KP focused on writing the core
- OSP: Covers development kernal and another O.S component like shell, user libraries, utilities.

—— x o ——

O.S ~~operation~~ operative modes (1/2)

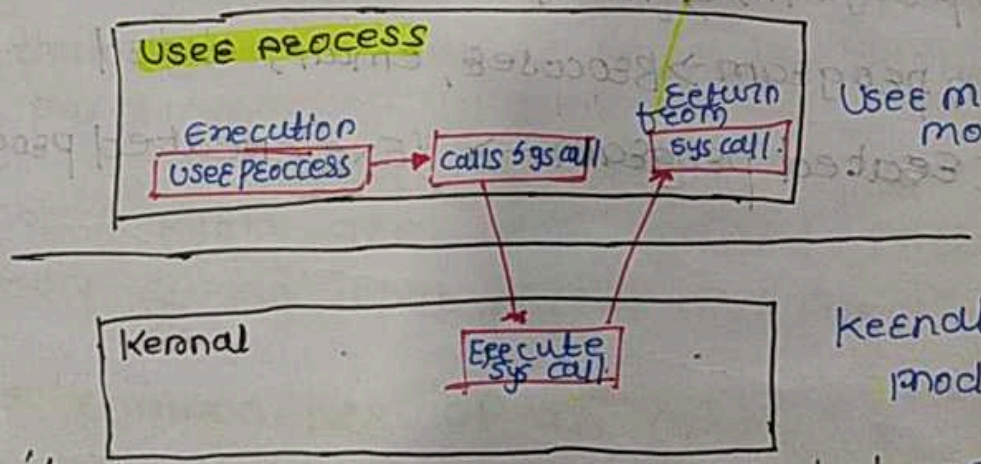
Operating system performs a two operations

i] User Mode

- Used for running user application.
- control interface between user and system
- In this mode application has less privilledge to access system resource

ii] kernal Mode

- Used by O.S itself.
- Controls system and hardware resource.
- Allows Full access to system memory and hardware, as it need to perform critical operations.



Mode switching :-

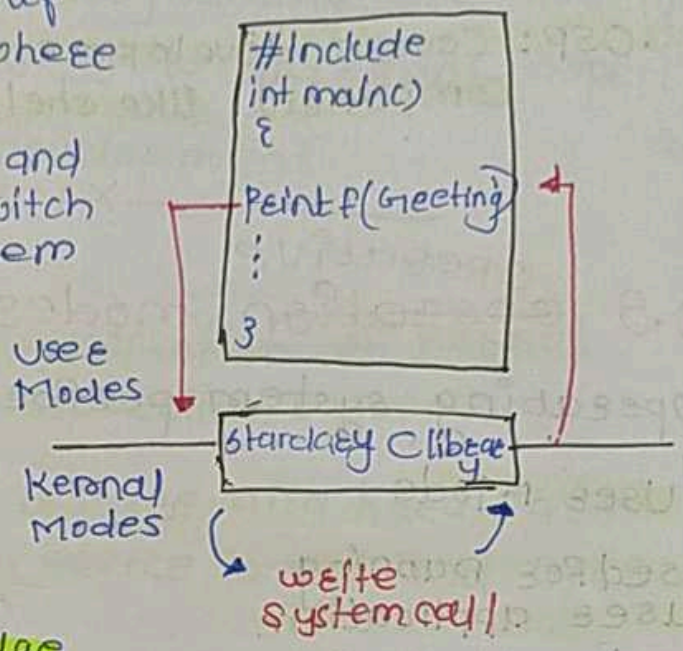
- The OS can switch between user mode and kernal mode.
- This switching is achieved through system calls.

- When a user process requires access to resource that are only accessible in kernal mode it triggers a system calls, temporarily switching to kernal mode. Once the task is completed, it return to user mode.

Q.5 Operations (operative Mode 2/2)

System call illustratestion for C's printf.

- When a program calls printf it operate in user mode where it get access to sys h/w
- The std lib intercept this call and translate into system calls switch to kernal mode to access system resources.
- this sys call enable os to handal output on a display once operation complets control return to user mode.



- Sys call are it acts bridge between system resource and user interfaces.

Os componets:-

1. Kernal → Runs all time in computer.
2. System program → file system stack, shells, N/W cards.
3. Application program → Browser, email, word
4. User created program → "Exe created program.

—: O.S Componets :-

The components of an O.S are foundation elements that enable it to manage hardware and software resources provide an interface for user, and run application.

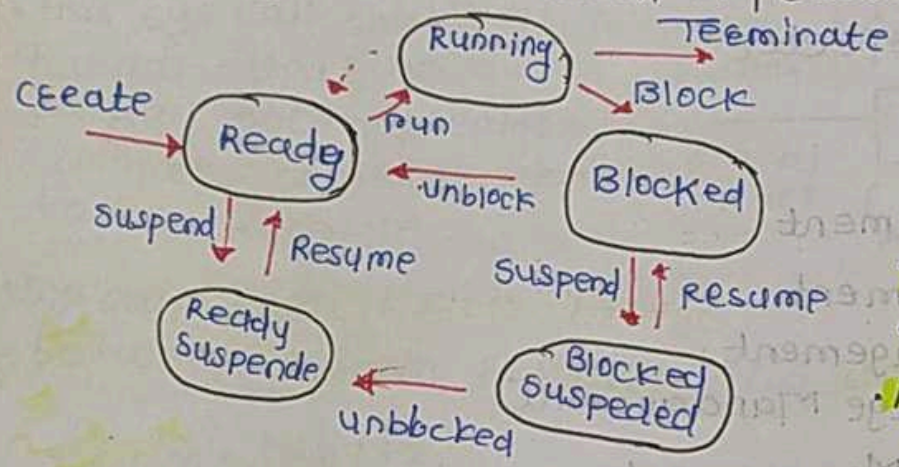
Impoetant componets of O.system :-

1. Process Management
2. Files Management
3. Command Interpreter
4. System calls
5. Signals
6. Network Management
7. Security Management
8. I/O Device Management
9. Secondary storage Management
10. Main Memory Management.

- An operating system is large and complex system and that can be created by only parting into small parts
- The above parts are well defined parts of system carefully during input, output and functions.
- These components of an operating system play a key role to make variety of computer system parts to work together.

Peocess Management:-

- A peocess is peogeam in enecution.
- A peocess is a peogeam that peogeam loaded in main memoey. it also needed resouces including CPU, time, memoey, I/o and files.
- So peocess Management componets manages multiple peocess simultaniously running on O.S
- The peogeam enecution in peocess:-



- When a peocess ceeate will peeform follow activity
- Ceeate, Run, eneate Terminate, resume
- Swith among pocess.
- Allocate de-allcate resoueess

- A peocess can be suspend teeminate temposily and enecution of othee peocess take up.
- befoee suspendend details save in peocess table.

O.S suppoet two peocess:-

i] System call:- Used to ceate a new peocess

ii] Kill :- Used to delete enecuting peocess.

- A peocess create multiple chiled peocesses.
- Peocess can communicate among themselve eithee using shared memoey and mebsage passing technique.

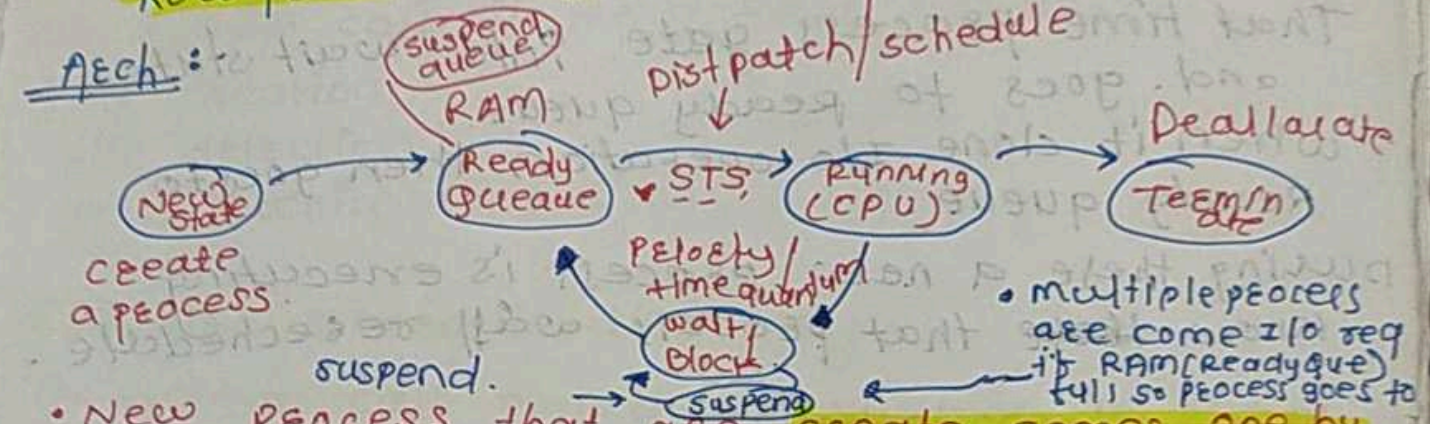
Keenal role:- Manages the ceeation, schedulling, content switching,

CPU:- Peovides usee-level utilites to monitoe and conteol peocess.

Process Life Cycle / Process states

Process states: Its model for user to understand how process work.

Arch:



- New process that are create comes one by one in Ready state or Queue.
- From new state (where process create) from there to ready queue the LTS (Longe term sched) are responsible to take process from new state to Ready queue.
- when the process comes from Ready queue to CPU know as Dispatch/scheduled. know CPU can run this then terminate. (~~Dispatch~~) (Deallocation).
- from Ready queue to Running CPU its change the state only.
- If the running state is process running. If suddenly an process come high priority or any RR scedulling Algo that time. the running process is comes to Ready queue and wait for that process to complete. this is done by STS (short term schedullers).

I/o Request durring Running State.

So when some process is executing but it requires any I/o files, or access h/w means it request for I/o operation.
That time process gate into wait state and goes to Ready queue.
when it done I/o operation then goes to Ready queue.
During these a new process is executing know ~~that~~ that process will reschedule.

Medium term scheduler.

If our ram (Ready queue gets full because of I/o operation the process goes to suspended wait state know ~~the~~ these in a secondary memory.
This process is done by Medium term scheduler.

Case when our Ready Queue get full and ~~one~~ a sudden kernel High priority task Came that time the that process goes to suspend Ready state.

Files Management:-

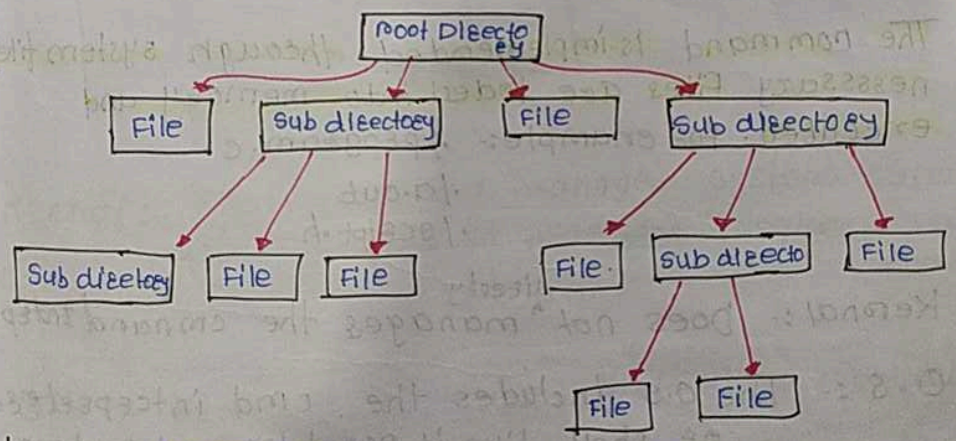
- Files are used for long-term storage.
- Every operating system provides file management services and OS also provides a system calls for file management

The system call file managemet includes:-

- File creation
- File deletion
- Read & write operations.

- Files are stored in directory
- An system call provide a service to put a file in a directory or remove a file from a directory

- Below is show Hierarchical files structure of directory:-



Kernal:- Manages operations(opening, reading, writing, close) interact with file system and storage.

OS:- Provides a interface for file management such as file enployees, permission settings.

Command Interpreter:-

- There are several ways for user interact with the operating system

  The interfaces provided for users:-

  i] GUI:- Graphical user interface

  ii] CLI:- Command line interface.

- Cmd line interpreter provides a CLI.
- It allows the user to enter ~~cmd~~ command on cmd prompt
- & the cmmad Interpreter accpt & execute the commands entered by user.

The command ~~line~~ to erected are implemented two ways:-

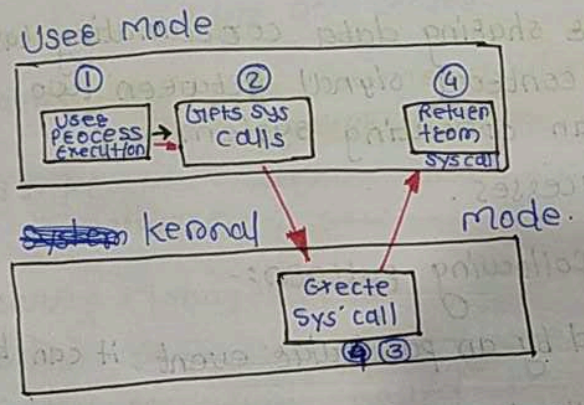
- The commad Interpreter itself contains code to be executed.
- The command is implemended through system file nessssary files are loded into memory and executed. For example:- ./program.c
  ./a.out
  ./script.h.

- Kernal:- Does not directly manages the cmnand Interpreter
- O.S:- The O.S includes the cmd interpreter or shell like it provides a interface a application prompt for user like example:-
  - bash:- in Linum
  - Command prompt:- in windows.

System Calls :-

- System calls provides an interface to the services made by an operating system.
- The user interact with the operating system prams through calls.
- The system calls normally made available as library functions in high-level language C, Python, Java.
- Different H/W and S/W services can be available through system calls.

Working of system calls:-



- sys call available for:-
  - Process Managemt
  - Memory Manag.
  - File Operation
  - I/O operation

- Kernal :- Provides and manages system calls. allowing user to ~~manages~~ request service from kernal.

- O.S :- Provides libraries that allow applications to make sys calls more easily.

Signals :-

- Signals are used in operating systems to notify a process that a particular event has occureded.
- Sigls are also an s/w and H/W interupts that suspent the current execution of a task
- Signal are also used for inter-process communication

Inter process communication:-

- (IPC) set of methods that allow ~~these~~ processes to communicate with each other and syncronis their actions.
- IPC essenial for sharing data, coordinating tasks are sending a control signal between two processes in an operating system.
- between two processes.

- A signal ~~flow~~ follow following patteen:-

- An occourance generated by an particular event it can be clicking mouse
- A generate signal delived message.
- once delived signal must be handled.
- A signal can be syncronisy and Asyronyse which handle by default handler or by user defined handler.

Kernal :- Message sending and Reciving signals. which are notification sent to processes to indicate event

OS :- The O.S provides higher level tools and libraries for managing signals, often allowing developers to define custom signals handler in their application.

Network Management:-

- In today digital world, the complexity of Networks and services has created modern challenges for IT professionals and users.
- Network Management is set of process and a procedure that help organization to optimize their network
- N/w management systems is a software application that provides Network Administrator with information on componets in their networks.

s/w:- i] Solar Winds Network performance monitor.

ii] Nagios.

iii] Cisco DNS center.

Kernal :- Manages low-level N/w operation such as a data packet routing, socket communication.

OS :- Provide Untilities and ~~operations~~ interfaces for networking, such as N/w config tools.

Security Management:-

Various process where user changes, files, memory resources that should have an authorization from the ~~OS~~ O.S

Kernal:- LL-security task:. enforing permission, us authentic.

O.S :- User level tool :- Firewall, antivirus.

I/O Management::

This component provides buffer cache and General Device driver code that allow ~~a~~ system to manages ~~and~~ hardware and main memory and h/w device connect to it.

Kernal :- Handle direct ~~m~~ communication with I/O devices through device drivers

OS:- Provide user interface to interact with I/O devices.

TAT - Turn Around Time

- Total time taken for a process to complete its execution.

WT :- Waiting Time

- The total time a process spends waiting in ready queue.

-: Operating System interfaces :-

operating system support different ways to interact with with user.

1/ Categorized into two ways:-

i/ Non-Graphical Interaction (CLI - Command line Interface).

- User types commands in the terminal or shell to interact with the O.S

Example :- Linux shells (Bash, Zsh ..).

ii/ Graphical Interaction: (GUI).

- User interact with the O.S using graphical iconic button and menu typically using mouse & keyboard.
- Joystic Interaction.

Example :- Windows 7/8/10 ..

-: The Role O.S in computing :-

- Enhance user interaction with h/w
- Effient utilization of H/R.

-: Operating System :-
Views And Viewpoints...

. Four Main perspectives:-

1./ Classical Views:-

i/ Resource Management: A O.S mamages all componets.

ii/ Master Control Program:- Control Exec of control User programes, Manages I/o devices.

iii/ Command Executor:- provide GUI Interface.

2/. Modern View:- Cloud Era:-

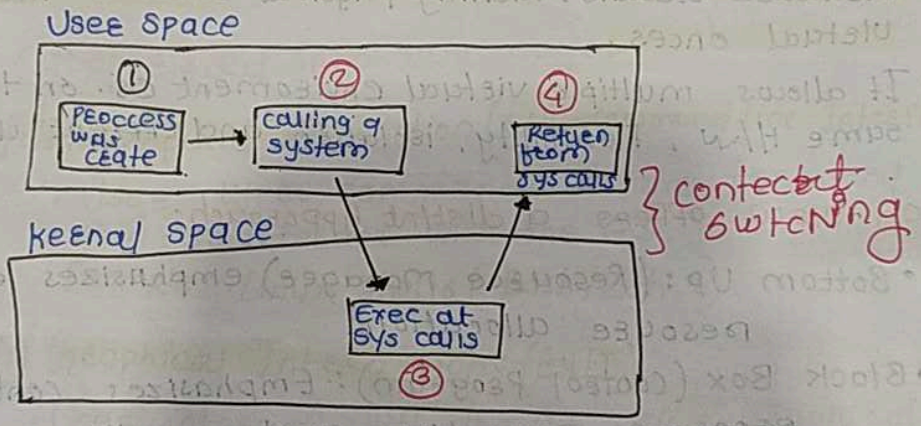
- OS have a high level virtual environment abstracting hardware details, turning physical resource into virtual onces.
- It allows multiple virtual environment OS. on the same H/W, flexibility, isolation, and effient use

Each view offers a distint Approach:-

- Bottom Up: (Resource Managee) emphasizes efficent resource allocation.
- Black Box (Control Program):- Emphasizes control ~~program~~ execution and security.
- Top-Down (Command Executor):- emphasises user interaction.
- Virtualisation:- Enhance flexibility and abstraction making H/w details transpheant to user.

-: System calls:-

- Sytem programming start and ends with system calls
- Sytem call is function invocations made from user space text editor, game so-on into a kernal in a order to request some services or resources from the operating system.
- Working of system calls:-
  - A system-call generate by user-space to kernal space for that a particular process will be executed.



- Used in a
  - i) Process Management.
  - ii) Files Management
  - iii) I/O operation Management.
  - iv) Memory Management.
  - v) communication.

Features of system call:-

*i] Interface :-

- system calls offers a well defined API that program use to access kernal service. This interface allow programs to request services such as file operation, process control and more.

*ii] Protection :-

- System calls allows access to priviledge operations that normal user-mode program cannot perform directly.
- This include accessing H/w. handing device operation and manipulating protected area in memory.

*iii] kernal mode :-

- When a system call is made, the user mode program temporary switches to kernal mode to perform requested operation.

*iv] Context switch :-

- invoking a system call requires a context switch between user mode to kernal mode

*v] Error Handling :-

- System calls include machanism for error handing by returning specific error codes.
- These codes inform the calling process if an operation is failed and provide information on the nature of issue
- which aids in debugging and handling exception.

*vi] Synchronization

- System calls allow/help to synchronize access to shared resources across multiple resource & Threads.

Linux System Calls for Process Management....

1. Creating a New Process....

Purpose: "Fork()" is used for create a new process. refers a child Process.

- This new process is new-process is almost exact duplicate of the calling (or "parent") process.

How it Works:-

- When "Fork()" is called, the ~~pro~~ operating system creates a new process with its unique Process ID (PID).
- Child process inherits most of the parents properties including file descriptor, environment variables and code.

Pid:- Process ID.

- It is refers to process ID in computing which is a unique identifier assigned by the operating systems to each active process.
- Every process has there unique process IDs
- There are some distinctions, such as a different PID and most cases, a distinct memory space

Returns:-

GetPid() - parent process ID
PID - child process ID

Fork() Returns:-

1. In the parent Process:-
   - when "Fork()" is called the parent Process recive the process ID of newly created child process as a ~~child~~ return value.
   - This allow parent to know child PID which can be usefull for parent to interact with or monitor chiled process.
   - For example:

```
pid_t pid = Fork();
  if (pid >0) {
    pf(" This is parent process. child PID : %d/n", pid);
  }.
```

2. In the child Process:-
   - when "Fork()" called it attempt to create a new process it will return pid of chid. if the process is a child process then it return a 0.
   - for example:-

```
pid_t pid = Fork();
if (pid==0) {
    printf(" This is the child process:).
}
```

3. If the Fork() is Failed...
   - if the Fork() call fails, it return -1 in the parent process. and no child process is created.

For example:-

```
pid_t pid = Fork();
if (pid == -1) {
    perror(" Fork faild");
}
```

2. Replacing current process.

Purpose:- "exec()" is a ~~fmai~~ family of function

- used to replace a current process with new process.
- This means that after acalling "exec()" the process stop executing the current ~~process~~ program and run new one.

How it Works:-

- When exec is invoke, the operating system loads the current program into a current process memory space
- overwriting executing code, data, and stack segment.
- "exec()" does not return to calling program unless there is error.

Returns:-

- If "exec()" is succsesfull. it does not return because the current process image is replaced by the new program.
- If there is an error "exec()" return "-1" and set "errno" to indicate the error.

Example Usage:-

```
char *argv[] = {"ls", "-l", NULL};
execv{"ls", args};
```

3. Synchronizing with child processes.

Purpose :- "wait()" and related calls (e.g., waitpid()) allow a parent process to wait for a child process to terminate, capturing the exits status and freeing resource associated with the child.

How to Work:-

- when wait() and related calls (e.g. waitpid()), it blocks the parent process until one of its child process exits.
- If child process is already terminated, wait return immidiately with the child's exit status.

Returns:-

- wait() return the PID of terminated child.
- If error occurs (e.g if there is no child process) it return -1.

Example:-

```
pid_t pid = Fork();
if (pid == 0) {
        // child process
            exit (0);
    }
else {
        // parent process wait for child to terminat
    wait (NULL);
}.
```

Libeaeys Used in Peocess Management:- # Headee files:-

i) <unistd.h>:

- Peovides decleeations foe many standaed system calls, including peocess-pelated once such as "Foek()", "_exit()", and "exec()".
- Also includes functions foe getting the peocess ID "(getpid())", paeent peocessid "getppid()".

ii) <sys/types.h>

- Defines data types used in system calls "pid_t (peoccess ID type) which is essentials foe handling peocess.

iii) <sys/wait.h>

- Peovides decleeation foe peocess conteol specially foe handling child peocess.

Impoet thing:-

1. The oedee in which the paeent and child peocesses execute is determined by the opeeating systems scadulee.

- So theee is no gueanteed oedee foe which peocess will run fiest aftee "foek()" called.

- C program that demonstrate process creation and management using "fork()".
- This program will create a child process, and both the parent and child process will print

Programme :-

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //for fork(), getpid(), getppid()
#include <sys/wait.h> //for wait()

int main () {
    //creating a new process using fork.
  pid_t pid = fork();
  if (pid < 0) {
    //Error handling for fork failure.
    perror("fork failed");
      exit (1);
  }.
  else if (pid == 0){
    // child process block
    printf("This is the child process.\n");
    printf(" child PID : %d /n", getpid());
    printf("Parent PID : %d\n", getppid());
    // Do some task in ~~task~~ in the child process
    printf("child process is running its own code...").
  } else { // Parent process block
        printf("This is process. \n");
        printf("ParentPID : %d", getpid()); //PPID
        print("child PID from parent: %d \n" pid);
    //Wait for the child process to complete
        wait (NULL); printf("child process compled"); }
}.
```

"Fork" Process creation in parent Process.

Calculating: where n is no. of fork

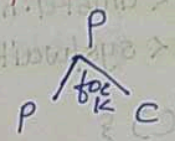
Parent Process: 2^n

calculating child process : $2^n - 1$

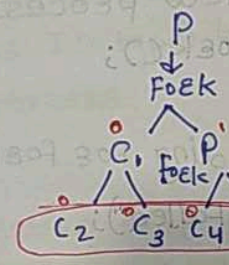
i]
```
main()
{
    fork()
    printf("Helloworld");
}
```

This program execute 2 time

p - parent
c - child



ii]
```
main()
{ fork();
  fork();
  Pf("Hello");
}
```

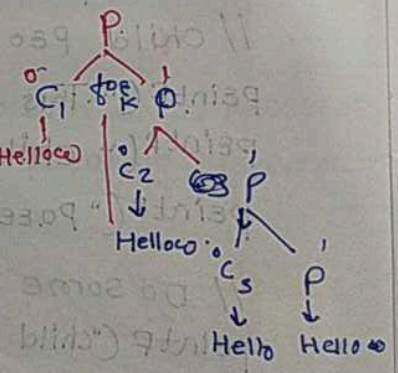


Pprocess - 1
c-process - 7.

Total process = 4

iii]
```
main()
{
    if (fork() && fork())
        fork();
    Pf("Hello");
}
```

4 Process



GNU Make Utilites...

GNU Make:-

- GNU make is build automation tool that control the generation of executable and other files from a program's source files.

Terminology:-

1. Target:-
   - A target is an output file that Make is a instructed to build
   - In makefile rule, the target appers at a beginning of line before the colon (:)
2. Recipe:-
   - A recipe is set of command to build the target.
   - Recipe apper independed (usuall by tab) directly below target. is out-of-data.
3. Lables:-
   - Labels are group of tasks in Makefile.

$@ : The target name.

$< : The first prerequisite(dependency).

$^ : All prerequsites.

- No need to recomple source file that not to be modified. those have change they compiles.

Scenario :-

- We write a program and divided into hundred of .c programs files and some header files.
- To make executable we need to compile those hundreds source files to create hundred obj files then we need to link those files to final executable.
- What happens if we make changes to one of these files.
  - if we are changes the program then we need to first clean then make.
  - (a) Recomple all files & then link all of them.
  - (b) Recomple the file which has changed and then link
- Make utility uses this dependency specification in the make file and the time when various components were modified.
- In order to minimize the compilation.

Structure of Make file.

target : dependency1 ... dependency n.
<tab> command

Name of executable to be build

Name of files on which target dependents.

shell cmd to create the target from dependencies.

- A every rule describes the dependency
- Advantage of Make utility :-
- Makes management of large s/w projects with multiple source files easy

System Call : File.

- File operations are fundamental to the most programs. They are typically performed using system calls which are function provided by kernal to interact with system resources.
- System calls related to file operations provide an interface for interacting with the filesystem.
- These system calls allow to perform basic and advanced Operation directly, providing flexibility and control over file handling.
- In the system calls (syscalls) are used by user programs to interact with kernal allowing them to perform operation on files, manage processes and more.

The file operations:-

1. open()
2. read()
3. write()
4. close().

1. open()

- purpose:- open() is used to either open an existing file or create a new file if it doesn't already exist.
- It establish a connection between a file and file descriptor.
- File descriptor - is unique identifier the kernel assigns to manage files.

Syntax:-

```
int open(const char *pathname, int flags, mode_t mode)
```

- pathname:-
  - This is the path of the file to be opened or create.
- Flags:-
  - This integer parameter controls how file should be opened, and have an option to open a file.
  - Common flags include:-
    - O_RDONLY:- open file in read-mode.
    - O_WRONLY:- open file in write only mode.
    - O_RDWR:- open file in Read-write mode.
    - ~~O_RD~~
    - O_CREAT: (Create a file if doent exist).
      This parameter required third mode.
    - O_APPEND:- Enable append mode.
      - Any data written in file is automatically appended to the end of file.
    - O_SYNC:- synchronous writes.
      - This means all data written to file is immediate flush to disk before "write()" returns.
    - providing a gurente that the data is physically stored on the device.

File Permission Mode:-

- In linux, file permission are controlled by/through modes, which define what type of access (read, write, execute) are allowed.
- This access are for a user and we have categories of user:-
  - File owner (user).
  - Files group
  - others (everyone).

File Permissions:- Mode:-

i. User (OWNER) permission.

- These permissions apply to owner of the file.
  - S_IRUSR: User has read permission
  - S_IWUSR: -||- Write -||-
  - S_IXUSR: -||- Execute -||-
  - S_IRWXU: User has read, write and Execute -||-

ii. GROUP Permissions:-

- S_IRGRP: Group has read permission.
- S_IWGRP: Group has write permission.
- S_IXGRP: Group has execute permission
- S_IRWXG: Group has read, write and execute permission.

iii/ Other (Everyone) Permission:-

S_IROTH: other has read permission
S_IWOTH: -||- write -||-
S_IXOTH: -||- Execute -||-
S_IRWXO: -||- read, write, execute -||-.

Linux file system permission:-

-rwx-rwx-rwx-

user
group
other

numerical form permission.

- read - 4
- write - 2
- Execute - 1

0700 - user have all per.

Returns:-

. The open() system call return an integer that serve as an file descriptor and indicates that success and failure of the operation.

- Non Negative No.
  - If "open()" successed then it return positive integer.
- -1 on failure.
  - If "open()" is failed then it return -1.

2. read ( )

Purpose: read it used to read data from memory.

- This system call reads data from other system files like pipes, sokets and device files.

Syntax:-

ssize_t read(int fd, void *buf, size_t count);

fd :- The file descriptor returned by open();

buf :- A pointer to buffer where the data will be stored.

count :- The maximum noof bytes to read.

Return value:-

- "read()" returns the no of bytes actually read which may be less than count.

3. write( ):

- Purpose:- write is used to send a data from a buffer to a file on a filesystem.
- Syntax:-

ssize_t write(int fd, const void *buff, size_t count);

fd :- The descriptor of the file to which data will ~~return~~ written.

buff :- A pointer to the buff containing data to write

count :- The number of bytes to write from buff to file.

Return values:-

"write()" return the no of bytes actually written.

4. close ()

- purpose (): "close()" is used to relate a file descriptor and finalize any outstanding operations
  - its prevent/important for freeing up a system resource leaks.
- Syntax:-

  int close (int fd);

  - fd : The file descriptor to close
- Return value:-
  - On success, close return 0.
  - If an error, occur it return -1.

Error Handling in O.S

1. Return Codes in functions and system calls.

- System calls:-
  - It interact with the o.s.
  - Example of system calls are "open()", "read()" "write()" and "close()".
  - These used for file management. interacting with H/W device.
- Return Codes:-
  - Functions, including system calls, usually indicates success or failure by returing an integer value.
    - A return 0 indicate success.
    - A return -1 indicate failure.

2. Globle variable errno.

Purpose:- In a C programming, if a function and system call fails it does not provide detailed information about an error.

- Insted, it return -1. and error details

How errno works:-

- When a function calls (e.g. open(), read() it return -1 or NULL then errno sets to indicate the type of error.

Errno Location:-

- errno defined in <errno.h> header file

Type:- is globle variable of type int. it value is set by system calls and library function when it occurs, then it accessed by special error code.

Common error code in errno:-

- ENOENT: No such file or directory (error code 2)
- EACCES: Permission denied. (error code 13)
- ENOMEM: Out of memory (error code 12).

5. Function to print Error information.

. Two standard functions are available to help display the error information contained in errno:

- perror() is a function from the stdio.h and error.h libraries that prints a description of last error that occured.
- strerror():
  - strerror(): return a string that describes error associated with a given error number.

Ex:-

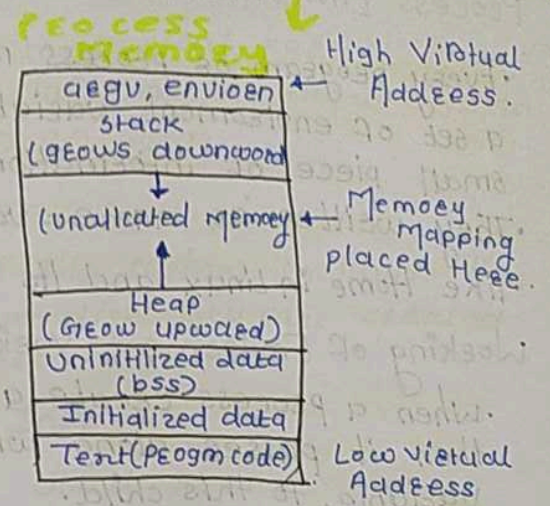
perror:- perror("File open error");

Processes And Threads..

- Process:-
  - A process is an O.S is an instance of program execution
- Key Componets of process:-
  - Memory Address Space:-
    - Each process has its own private memory space that includes segments like stack, heap and data which are inaccessible by other process unless it to share.
    - This isolation of memory managed by Memory Management Unit.
  - Thread of Execution:-
    - A process has at least one thread of execution, often called the main thread which Execute instruction.
- Process Resources:-
  - Memory Resource:
    - Include the stack (For function call & Variable alloction).
    - heap (Dynamic Memory Allocation) and other are memory (like data and code segment).
  - File Descriptor:-
    - Process May hold file descriptor or handles to other resources (like N/w, Connections) Managed by O.S.

Process Memory layout:-

- In a typical process, memory was divided into several segments.
- Each process segment has different purpose and is organised from low to high virtual Address.



- Text segment:
  - This segment contain Machine-code instruction that make program executable code.
- Initilsed data segment
  - store the global and static variables that are initilized by program.
- Uninililized segment:
  - contains a global and static variable that are uninitsilized by program.
- Heap segment:
  - Heap is dynamic memory area where memory can be allocated and freed during the program execution
  - when program requist memory like malloc it comes to heap.
  - Heap goes upward as memory address allocated. Hmemory
- Uninitilized Memory:
  - This section lies between stack & Heap serves buffer to prevent them from overlapping immeditely.
- stack : used to store local variable function call info.
- Argv, environ:- Are argument pass through program like in cmd line argy argV and variable.

Process Environment:-

- Every program or process running on system has a set of environment variables. these are like small piece of information attached to process.
- They written as name = value
- like Home in Linux and its hold value or info.

Working of Environment variable:-

- When a process create a new process (child Pro). then it passes along with its environment variable to this child.
- Child have same variable number as process P.

Why Environment variable are usefull:-

- Program can look at these environment variable to decides how to behave without asking for additional input.
- That means if we have a one specified variable and that variable contain a info about that type of variable is know as a process environment.

Like Example:

USER : Variable tells a program who is USER.

Home :- —||— where the user's home folder is.

PATH :-||- lists places where should looks for programs you want to run.

One-Way Communication

parent → child.

Process /proc file system in Linux:-

- "/proc" file system is special part of linux system that provide information about system process that are run on ram.

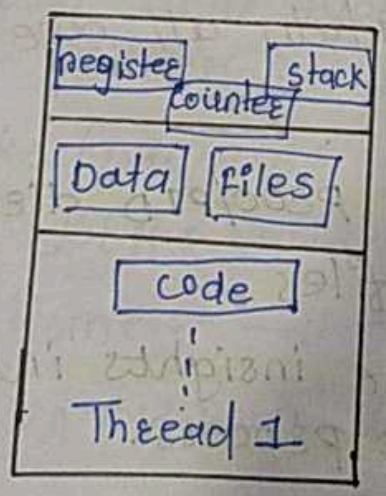
1. Pseudo-file system:
   - /proc is a virtual file system in linux the file does not exist on disk they are created dynamically on system ram.
2. It Exposes kernal Info.
   - /proc provide info about kernals process Hardware and system setting.
   - Include memory usage, CPU info.
3. Files and Directories
   - The /proc directory is organized with subdirector and files. These all directory represent sys info.
4. Files not Exist.
   - In /proc the files are virtual they are not store on disk.

- Note: All Files that /proc contain all are only a read-only.
- Every running process has proc/PID directory contains subdirector and files.
- proc/pid provides a real-time insights into what happing with a specific process.
- Tools like top, ps and htop is pull data from this director and display process info.

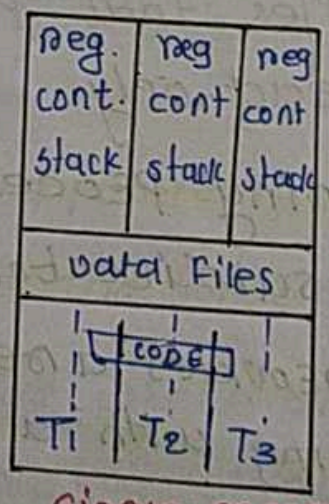
-: Thread :-

- A thread is a single stream in process having a same properties as a process have that why thread also called lightweight process.
- They are executing one after other
- Define:-
  - Threads are small units of computer program or process that can be run independely
  - These allow programe to perform a multiple task at same time like having a different types of process in a process.
  - These make program responsive and efficent.
- Each thread has its:-
  - A program counter
  - A register set
  - A stack space.
- Each thread can share Data and files.

Working of Single and Multi-thread:-



Single process P with single Thread



Single process P with multiple thread.

- Each thread belongs to exactly one process and no thread exist outside of a process.
- Each thread represent flow of control.

Need of Threads:-

- It create a new thread process take less time.
- Thread can share data common.
- context switching is faster when working with threads.

Similarity in between Threads and Process.

- Both create a children.
- Both scadulel by the O.S. based on scaduling Algo OS gives or assigne a CPU time to execute.
- Both have their own execution context.

Difference between threads & Process.

- Resource
  - Process have there own address space and resource it have we not communicate resource so coe use IPC
  - where threads share memory and resource with program that create them.
- Scheduling:
  - Process are scadule to use use a processor.
  - Threads are schadule to use the processor by OS. or program itself.
- Creation
  - Process managed by O.S.
  - Program or process are managed threads.

What threads shares :-

Threads in process share :-

- PID (process ID)
- Parent Process ID
- UID (User ID) and Group ID (GID)
- Controlling and terminating
- File descriptor and locks.

Distinct for each thread :-

Each thread have unique thread ID and not share.

- Tread ID
- Signal Mask
- errno variable : isspefic to each thread.
- RT (Real time) scadwing policys.

Posix thread API :-

- Posix :- It is a standards for maintaing compability between operating system.
- POSIX :- portable Operating System Interface.
- It ~~consists~~ define consistent API and enviroment to ensure compability of (Unix like system).

APIs for Create/open.

Function prototype:-

```
#include <pthread.h>
~~int pthread_create(pthread_t *thread, const pthread_att~~
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Function detailed explanation:-

1. Pthread_t *thread

- pthread_t * is a pointer to a variable of type pthread_t: it used to store unique identifier (ID) of thread.

why we use it:

- The thread id used to track and manage thread.
- pthread_t is defined in <pthread.h> header file.

Example:-

```
pthread_t thread;
pthread_create(&thread; Null, thread function, NULL);
```

2. Const pthread_attr_t *attr

- const pthread_attr_t * is a pointer to a variable that is pthread_attr_t structure that specifies attributes (properties) newly created thread.
- eg (stack size, scadwling policy, detach state)
- If we pass Null the default thread attributes used.

Why we use.

- For customization:
  This parameter allows specifys or costomise the attributes such as size of stack its scadwling.

3. Void *(*start_routine)(void *)

- Void *(*start_routine)(void *) this is a pointer to a function that new thread will be execute when it starts.

The function must match will following signature:

```
void *start_routine(void *arg);
```

where as:

- arg: The argument passed by function when the thread is starts (typically used to pass data)
- Return type: should be return (void * pointer.

why it use:

- Entry point of thread. When a new thread is created it begins execution by calling this function.
- These allows define what thread does (its task).

where it available:

- A function start_routine can be difine in our program. It defines what work new threads will perform.

Example:

```
void *thread_function (void *arg){

}
```

4. void *arg

- This argument pased to a function.
- The void* arg is allow flexibility because it can be used to any data type
- The argument can passed info like integer, structure to the new_thread.

why it use:

- Data passing:
  - This allow to pass any kind of data to the thread.

— xox ~~Return V~~ — xox —

Return Value of Pthread:

- The retuen value of p_thread_create is 0 on success.
- and non-zero for ~~and any~~ failure.

why we use:

- Error Handling:
  - Determine the thread was create or not.

Example:

```
if (pthread_create(&thread, NULL, threafun, NULL)
                                    != 0){
    perror
```

Termination options:-

- There are multiple ways in which a thread can terminate.

1. pthread_exit (void *retval).

- The thread explicitly terminate itself. it means thread calling a function with a purpose of terminating or ending its execution.
- A thread taking an active role to in stopping its own process, rather than being being terminated by external event.
- For example: a thread call exit function to explisity stop itself, signling that it finish with its task should no longer continou running.

2. int pthreat_cancel (pthread_t ~~cancel~~ thread).

- This thread requests cancellations of specific thread.
- Behavier:
  - Its sends a cancellation request to target thread.
  - The targeted thread cancel/terminated imidetly or at a specific cancelation point. depending on a setting.
  - If the thread has cancellation disabled, it will not terminated until cancelation is available point is reach.

Use case:-

- when we want to stop a specific thread from a executing like in scenario a shutting down a process or clenning up after errors.

3. int pthread_t join(pthread_t thread, void **retval);

~~This function waits for a thread terminated thread's~~

- This function waits for a thread to terminate and optionally retrive its return value.

Parameters:-

- thread : The thread to wait for
- retval : A pointer to a variable where the terminated thread's return value (from pthread_t_exit() or start_routine) will be stored.

Behavior:-

- The calling thread is blocked untill the specified thread finishes.
- If the thread has already terminated, pthread return imidetly

Return value:

- 0: for success.
- Non-zero for errors.

Use case:

- To synchronised with termination of another thread and retrive any return value provided.

4. Return from start_routine.():

- when a ~~function~~ thread finishes its from start routine, it terminated.
- when the work of thread is done then it will be naturaly terminate the thread.

Behavior:

- Returning from start_routine means equivalat to calling pthread_exit(); with return value as the parameter.
- The return value of the function can be retrived by the thread that called pthread_join();

- Threads some part is explained in a after semaphore topic because it will be important to know about other thing.

Prabhakar

-: Creating a Process :-

- Process is created by using "fork()" call.
- #include <sys/types.h> // this header files define data types used for syscalls
  #include <unistd.h> // this header file provide access to POSIX sys API, which include function for process mang., file handling and other system level operations.

```
{
  pid_t  fork( );
}
```

- Child is a Exact copy of parent Process but a child process have a different address space.

-: Terminating a Process :-

- Process terminates 2 ways:-

  termimination — Voluntary
  — involuntary.

i] Voluntary Terminations:-

- A process terminates itself by calling the exit() functions.
- #include <stdlib.h>
- exit(0); // Voluntarily terminates, passing 0 as the return value.

ii] Involuntary Termination

- The process is terminated by an external signal such as:-

  1. SIGKILL (forcefully kills the process)
  2. SIGTERM (request terminations or ignored but can be handled by process).

SIGKILL and SIGTERM:

- They both are used to terminate the process.
- SIGKILL:- (Signal 9).
  - It is a Forced termination immediatly terminate the process without giving a chance to clean up.
  - The OS directly stops the process.
  - It used as a Last resort when a process does not respond to other signals.
    (Eg. when its hung or in an infinite Loop).
- SIGTERM (signal 15)
  - Graceful Termination.
    - Politely ask the process to terminates
    - It catches this signal, clean up resource, save data
  - The used case is performed standared process Termination.
  - If the process implements as a handler for a handler for sigterm
    It dedicates How to handle it. if
  - If no handler is defined then the process is terminated.

Return values of Terminating Process.

- This is exest/exit status of a processes used to indicates why the process is terminated it can be one of following :-

i] Argument to exit():
- when a process calls exit(value)
- The value is used as the exist status.

Examples:-
i] exit(0) // sccess
ii] exit(1) // An error occur.

ii] Signal Values:
- If the process is terminated by a signal the exist status indicates the signal number that caused termination.

collected by parents:-
- The parent process collect child process exit status using:

1. wait():
- Block the parent until one of its child processes terminates ex:- wait(&status);

2. waitpid():
- Allow parent to wait for a specific child process to terminates. pid(child pid, &status, 0);

Zombie Processess :-

1. what is zombie process :-

- A terminated child process that not had its exist status collected by parent becomes a zombies.

when the zombie process occurs :

- If parent doesn't call "wait()" or "waitpid()" the process remains a zombie.

How zombies cleared up?

- If pp terminate, the zombie child is adopted by the init process (PID 1). The init automatically collects the exit status and clean up the zombies.

-: Inter-process Communication :-

IPC is a critical concept in computing, it enable process-es to share data /exchange data and co-ordinates their activities

since processes are isolated with there own memory space, ipc mechanism are required to share info.

Modes of IPC:-

- There are Three mechnisms are available:-
  - *1] Message Passing
  - *2] share files & file locks.
  - *3] share Memory.
  - *4] socket
  - *5] Pipes and FIFOs.
  - *6] Message Queue
- These 3-modes mechnism categorised based on their communication style, Performance characteristic & synchro need.

Types of Process :-

1] Independent Process: It can not be affected or affect by the other process executing in the system.

2] Co-operating process: They can affect or affected by other process executing in system.

- Any process that shares data with other processes is co-operating process.

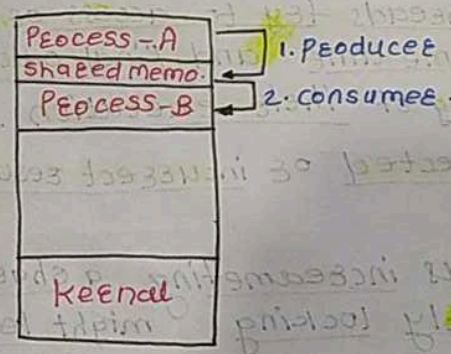
1] Shared Memory Method:-

- In a shared memory method is that we can take a memory space for sharing resources.

Example:

1. We have two process process A - producer, B - processes as consumer

- The producer - A produce some item and shared that on common memory space that space/location know as a buffer there the shared data will be stored.
- And from consumer needed data then it will be consumes from that common memory space/buffer.



- shared memory requires synchronization mechanisms (eg. semaphore, mutexes) to ensure data consistacy & avoid race condition. Next pg.
- A dedicated shared memory region is accessible by multiple process.

Advantage:-

1. Faster data exchange due to direct memory access.
2. Suitable for high - frequency or large value data sharing

Disadvantage:-

1. More complex to program.

Syncheonization Mechnisms:-

- It is a tool or technique used to co-ordinates the execution of processes or threads, ensure that shared resources (like memory, files and variable) are accessed safely in concurrent enviroment & correctly.

Why we need Syncheonization Mechnisms:-

In shared IPC, multiple process and threads have a simultanious accessed to common memory region without proper synchronization, the following issues are occurs:

1. Race Condition:-
   - When a two processes or threads try to access or modify shared data at same time and final outcome dependent on final timeing of their execution.
   - This can leads to unexpected or incurrect result.

Example:-
- Two thread simultanious increameting a shared counter without properly locking might leads to incurrect results.

2. Data curruption:-
   - A situation where multiple process or threads accessed and modify same data at the same time without proper corrdination this leads to invalid states in data breaking integrity of system

A mutex ensure the data proper

Process synchronization.

During interprocess communication cognative process that communicate between other process during various conflicts or states occurs that need to secure for that we use process synchronization.

In cognative process shares:

1. Globle variable
2. Memory [buffer].
3. Code
4. Resource
   - CPU
   - Printer.
   - memory section

Example:

$X = \cancel{5}\ 4$

P1	P2
x++;	x--
sleep(1);	sleep(2);
// share x	// share y
4	4

- Process Running in CPU
- P1 acquire 1st increement to 6
- Know process goes to sleep mode then P2 aquire X during this the variable get decroement
- The value of these are get change
- This occurs a <u>race condition</u>.

—— ||oxov ——

~~Race - Consumer problem:~~

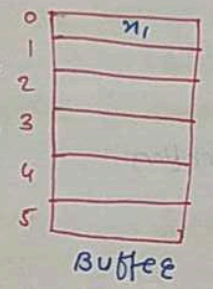
Race Condition prevent by:
- Mutex Lock.
- Semaphore.
- Spinlocks.
- rw-we lock.

Producer - Consumer Problem.

* Producer: who produce or share some resource data to buffer
* Consumer: Collects data from buffer
* buffer is shared by both producer & consumer.

Case 1: simple case

where producer produced 1 element in buffer and the consumer consume the data from there and counter must be used



* Buffer full overflow count If $3 < 5$

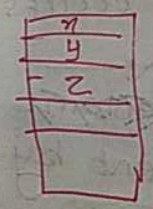
This is synchronised way case-1

if produ → buffer → C=1
co +1

if consumer → buff → C=0.
co −1

thes meachanism used when the case-1 used.

Case 2:-

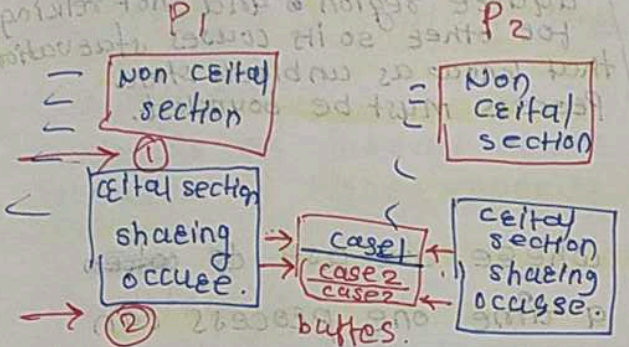


If process synchronisation not done then the Race condition can occure.

Critical sections:-

A part of program where some shared resource accessed by various process.

Example:-



$P_1 \rightarrow$ ① - we lock the process untile our work not done

② → unlock the process when the p work done.

then P_2 acquire shared Region and do whatever want to do.

Case 1 if both are trying to access at same time an fault is occurs so there is n data curruption occur.

Case 2

where $P_1 \rightarrow$ access change some data

send time $P_2 \rightarrow$ access change data.

mean both are changing data this occur a race condition heas ~~know~~ both process not get there desired output.

for that we use synconition.

—— X o X ——

Syncheonization must be follow following 4-Rules

1. Mutual Encusion.
2. peogeess
3. Bounded wait

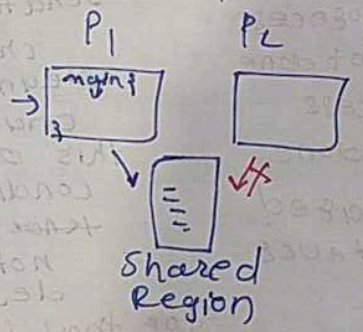
③ Bounded wait: mean every peocess get equal opportunity to get aresouce if the one peocess is contionsly aquiee region and not relking foe othee so its couses staevation that know as unblock state. Peocess must be bounded.

—//0x0//—

1. Mutual Encultion.

- Peogeamming concept wheee we use ~~to~~ ~~when~~ ~~the~~ foe ensuee that a time one peocess can only use a shaeed resouece. if this goal is done so its mean mutual Enclusion. is happen.

Example



if P_1 take shaeed region fieest then P_2 it not take access of shaeed region.

② Peogress.

—xoX—

Ie peogeess ensueing that if a one peocess wants to entee in shaee region foe finite amount of time these ensuee that the peogeess but some peocess aee blocking the othee peocess to get a shaee eegion then the peogeess is not good.

Peogeess must be followed.

3. Deadlocks :

- when a two system processes & threads become stack in a cycle of waiting for resource that are handled by each other.
- This situation causes the processes to halt because none of can proceed without obtanning the resource. it is waiting for.

4. Staevation :

- A process or thread never get access to a resource if higher-priority processes continously block it.

5. Consistency:

- Ensure that shared resource remain in consistant state during and after operations

—xox—

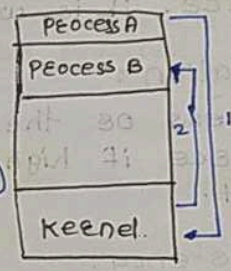
Common Syncheonization Mechnisms :-

1. Mutex : Used to lock critical section to prevent access.
2. Semaphores : allow a fixed no of thread to access a resource
3. Spinlocks : A lightweight lock where threads repeatedly check until lock become available
4. Monitors
5. Barriers
6. Condition Variable : Enable threads to wait for conditions to be true.

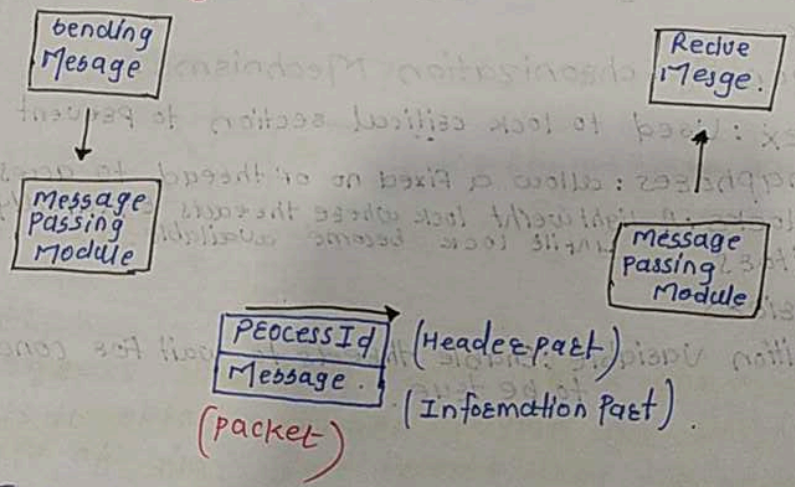
2] Message passing Method:

- In this method process communicate with each other without using a shared memory
- If two process wants to communicate with each other they proceed as follow:

- Establish communication link (if link exist, no need to establish it)
- start exchanging message using priority two priority:-
  - send (mesg, destination) or send (msg)
  - recive(mesg, host) or recive (msg).



Message Passing:



i] The standard message two part Header and info.

ii] Header:- storing the message type, destinationId source id, message id and control info.

- control info : contains what to do if runs out of buffer space, sequence Num, priority.

Communication happed by links

- In message passing, communication between a processes occures via links.

There are two methods:

i) Direct cummunication link.

ii) Indirect cummunication link.

Ipc Using Shared Files :-

- Ipc allows processes to communicate and syncheonize with each other. Using a share file is simple mechanism.
- That means a multiple processes (eg. reader & writer) access same file for reading or writting. that file used for as a common medium for communication between process.
- To prevent data from corrupting or conflicts, file locks are applied.
  - Ensure that only one process can write at time (Enclusive lock), while multiple processes can read concurrently (shared lock)
- After operation process release the lock and allowing others to access the file.

- Enclusize Lock : (F_LOCK)
  - Purpose: This lock is used when a process needs to ~~run~~ have enclusive access to the file, preventing any other processes from writing and reading to file.
  - Use case :- Its applied during write process, which needs to modify the file.
  - Behavior : while enclusive lock is held, no other process can aquire any kind of lock (niether read or write) on the file
    other processes attempting to acquire a lock ~~or~~ they blocked until the enclusive lock relea ed

• Shared Lock (F_RDLOCK):

Purpose: This lock allow multiple process to read the file at the same time, but prevents any othe processes from writting while lock is held.

Use case: Typically applied by reader process, which only need to read from the file and don't need to exclusive write access.

Behavier: Multiple reader processes can hold a shared lock simultanosly, but if writer wants to acquire an exclusive lock, it will block until all shared locks are released.

File Locking in Unix/Linux:-

The flock structure define in <fcntl.h> is used for specific file lock parameter.

Breakdown:

- l_type: specify type of lock are as follow:-
  - F_RDLOCK: shared lock for reading
  - F_WRLCK: ~~shared~~ Exclusive lock for writing
  - F_UNLCK: Unlock a lock.
- l_whence: The reference point for l_start.
- l_start: The starting offset of lock.
- l_len: The number of bytes to lock. Used to lock entire file.
- l_pid: The process ID holding the lock.

Impoetant points: about IPC file shaeing.

- To use fcntl(), we need to initilized flock steuctue peopeely as it holds nessaey paeameteeS Foe the lock.

steuctuee of flock:-

```
struct flock {
    short l_type; //Type of lock: F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence; //How to interpret l_start.
    off_t l_start; //starting offset for lock
    off_t l_len; //Number of bytes to lock.
    pid_t l_pid; //PID of process blocking our lock (set by F_GETLK)
};
```

I have weite a peogeom /CDAC/Embedded_OS/my EOS_CODES/IPC SHARED FIFE

—— xox ——

what is l_whence: This field in a steuct flock specifies how to interpreat the l_start field which defines the starting point of lock within file.

- This field deteemine refeeence point for calculating the starting offset (l_start) of the lock.

Possible Values for l_whence:

1. SEEK_SET:
   - The offset (l_start) is relative begnning of file.
   - l_start = 0 means lock start from begnning of the file.

2. SEEK_CUR:
   - The offset (l_start) is relative to current file possition (where the file descriptor currently points.)
   - Usefull when we working with the file pointer and want to lock a region starting from current position

3. SEEK_END:
   - The offset (l_start) is relative to end of the file.
   - This is usefull for if we want to lock a region at the end of file when appending data.

How l_whence Affect locking.

- Combination l_whence & l_start defines where the lock begins in the file.

1. Locking Entire file:
   - Set l_whence = SEEK_SET;
     l_start = 0;
     l_len = 0; // lock entire file.

2. Locking specific region:-
   - Set l_whence = SEEK_SET;
     l_start = 100; // position where lock start at byte 100;
     l_len = 50; // lock next 50 bytes.

3. Locking from current position:
   lseek(fd, 200, SEEK_SET); // Move fp* to byte 200
   - Set l_whence = SEEK_CUR;
     l_start = 0; // lock start current Posi 200 byte
     l_len = 100 // lock next 100 bytes.

starting from current fp*

4. Locking at Relative End: Lock Region at End of file.
   l_whence = SEEK_END
   l_start = -50; // before 50 bytes
   l_len = 50; // lock last 50 bytes.

L_pid : Helps to find out which process holds a conflicting lock on a file.

- We can check whether a file have lock or not or query cmds.
- So when we query cmds then l_pid will return the pid of the process holding the lock.

Scenario:

- when we have a two process running
  - Process A
  - Process B.
- Process A is running and its lock a file so for checking the Process A has successfully lock the file or not we use Process B with some query command.

Queys:-

1. GETLK (Get lock Information)

Why we use:-

- To check if region in file is lock by other process
- To know ~~how~~ who is holding lock (via pid).

- Scenario:
  - Process A (writer or locking process)
    - This process set lock (using SETLK or SETLKW) on a file or region of the file.
    - Process A is holding the lock (Not use L-pid its not query anything)
  - Process B (Reader or checking process).
    - This process want to check file or region lock.
    - It uses GETLK to check current lock status.
    - If Process A is holding a lock on the file or region, GETLK will tell Process 2 the PID of Process 1 through l_pid.

2. SETLK (Set lock without Waiting).

- It is used to set a lock on a ~~while~~ file without waiting if the file or region is already locked by another process.
- If lock can not obtained then set lock will return immediately with an error.
- key points:
  - Non-blocking: If region is already locked then SETLK does not wait it return error immediately
  - Type of lock: Exclusive & shared.
  - Scenario for SETLOK:-

1. Process A wants to write to file. It tries to get an exclusive lock.
   - If file lock by another process SETLOCK will fail imediately (no waiting).
2. Process B: shared lock
   - If Process A has file lock for writing SETLK will also fail immediately

If Process A is already using the ~~bao~~ ~~Process~~ file for writing, Process B can't read file. SETLOK ensure that Process B know immediatly that file is not available for reading and wont to have wait it simply fail and return an error.

Note:- I write a code for GETLK & SETLK and also for SETLKW that are in 'CDAC' folder path, CDAC/EM_OS/MYCODE.

3. SETLKW : SET Lock and wait.

- Similar to set lock SETLK.
- SETLKW will wait is file or region already locked by another process.
- It will block process untile the lock become available.

- Non-Blocking vs Blocking
  - SETLK : Fails immediatly if lock is not available.
  - SETLKW: Waits until lock is available.

- Behavier of Working:
  - Once the lock released by other process, SETLKW allows waiting process to acquire the lock and continue its operation.

Role of l_pid in SETLKW:

l_pid is not use is SETLKW because it will wait untile lock not get open.

Scenario:

- Process A has locked the file for waiting.
- Process B now want to access file but wait because Process A is holding the lock.

if ~~use~~ Process B uses SETLKW

- Process B will block (wait) until Process A release its lock
- Once Process A unlocks the file, Process B acquire the lock and proced with its operation.

IPC using Shared Memory:

- POSIX API for share Memory:
  - posix share memory is a mechanism where process share a region of memory using APIs.
  - APIs are designed to make the process easier and more flexible.

Posix APIs:-

1. shm_open( )
   - It is used to create or open an existing shared memory objects.
   - It return a file descriptor that can be used with other system calls ftruncate( ) and mmap( )

- Example:

```
int shm_fd = shm_open("/my_shm", O_CREAT | O_RDWR, 0666);
```

- /my_shm : Name of shared memory object.

Important point:-

- Why does it return a file Descriptor?
  - In POSIX system, shared memory treated as a files.
  - This is why shared memory represented in a /dev/shm/ directory.
  - The file descriptor allows OS to manage access to shared memory object just like files.

2. ftruncate(~~shm_fd, 4096~~);

- set the size of the shared memory objects.
- Example:

ftruncate(shm_fd, 4096);//set the size of fd 4kB (4096 bytes).

3. mmap() : memory mapped

munmap(): memory unmapped.

3.1 Mmap(): it maps the shared memory object into process address space so it can read write.

- Mapping: it create a connection between the shared memory objects (or files) and process memory address space. it behave like normal memory region.
- Why mapping needed:
- shared memory is kernal managed resource. without mapping our process cant directly access it. so mmap() makes share memory available.

3.2 munmapped:

- Unmapping is the process of disconnecting mapped memory region from program address space.
- It ensure proper cleanup and prevents memory spaces leaks.

Example:

```
void *shm_ptr = mmap(0, 4096, PROT_READ | PROT_WRIT, MAP_SHARED
                     shm_fd 0).

munmap(shm_ptr, 4096);
```

4. shm_unlink():

- Removes the shared memory object from the system, freeing the associated kernal resources.

Example:

```
shm_unlink("/shm_shm");
```

Important Notes:

1. Backing Files:
   - Shared memory is represented as file /dev/shm/.
   - This enables the kernal
   - This manage the kernal to memory manages the shared memory.
2. Concurrency Control:

Concurrency control:

multiple process/thread access same resource called as concurrency.

- Multiple process might read/write to same memory space simultaneously.
- This can lead race condition where outcome depends on order in which process and thread access shared memory.
- It can cause bug or unexpected behavior.

- To control access to share memory & ensure only one process can access the memory at a time.

So ~~th~~ thats why we use synconization primitive Semaphore and Mutex.

- Mutex: A mutex is locking mechanism that ensure only one process or thread access shared resource at a time.
  - If one process lock the process mutex, other must wait until mutex unlocked.
- Semaphore: It mechanism that idicates number of process allowed to access a particular resources at a given time.
  - A semaphore used for limiting access to multiple processes, not just one.

——— x o x ———

Creating / Opening and Closing Shared Memory

```
# <includ<sys/mman.h>
# include<sys/stat.h> // for mode constants
# include <fcntl.h> // for O_* constants.

int shm_open(const char *name, int oflag, mode_t mode);
```

- shm_open: create or open a shared memory object.

Parameter:
- name : Unique name for shared memory object (eg. /my_shm).
- oflag : operation Flags, Example:
  - O_Creat: Create a new shared memory object if not exist.
  - O_RDWR: open the memory for read/write access.
- mod : set permission (eg 0666 read/write by all).

int shm_unlink(const char *name);

- Deletes the shared memory object.
- The obj will be removed only when all process close it.
- This remove share memory object /my_shm.

Ex: shm_unlink("/my_shm");

2. Setting the size:

ftruncate()

```
#include <unistd.h>
# include<sys/types.h>
int ftruncate(int fd, off_t length);
```

- Set the size of shared memory obj.
- Shared memory start with 0, so you need to resize it before mapping.

Parameter:

① fd : The file descriptor return shm_open.

② length : The size (in bytes) to allocate.

Ex: ftruncate(shm_fd, 4096);

3. Memory Mapping / Unmapping:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
        off_t offset);
```

How it work:

- Maps the shared memory object into the process memory space
- Once mapped, processes can access shared memory as regular memory.

Parameters:

1. addr: Memory address (pass NULL to let the system chose).
2. length: size of memory to map.
3. prot: Memory protection flag.
   - PROT_READ: Allow reading.
   - PROT_WRITE: Allow writing.
4. flags: Mapping options:
   - MAP_SHARED: Memory changes are visible to all processes.

~~5. flags:~~

5. fd: File descriptor from shm_open.
6. offset: start mapping from this offset (usually 0).

```
void *shm_ptr = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                     MAP_SHARED, shm_fd, 0);
```

```
unmap( void *addr, size_t length);
```

- Unmaps the shared memory from the process memory space.
- After unmapping, the process can no longer access the memory.

Parameters:

- addr: The memory address returned by mmap.
- length: The size of mapped memory.

Example:

```
munmap( shm_ptr, 4096);
```

Semaphores Overview: — this used for file operation

Semaphore is used for synchronizinting priorities used to control access to shared resources. They are often ~~used~~ with shared memory for concurrency control.

Open/Create a semaphore:

```
#include <fcntl.h>      /* O_* consonts */
#include <sys/stat.h>   /* For mode constants */
#include <semaphore.h>  /* For semaphore functions */

sem_t *sem_open(const char *name, int oflag, mode_t
                                        mode,
                                unsigned int value);
```

Parameters:

- name: Name of semapore "/my,semaphore").
- oflag: Flags O_CREAT, O_RDWR.
- mode: permission bits (0666. r/w).
- value: Initial value for semaphore 1. for binary.

wait on/ Post a semaphore:

```
int sem_wait(sem_t *sem);
```

- Decrement (lock) the semaphore.
- If value is zero, it will block until another process increment it.

```
int sem_post(sem_t *sem);
```

- Increment (unlock) the semaphore allowing other process

-: Deadlocks :-

Semaphore.

- A synchro mechanism used by O.S. It ensure that multiple process or thread access resource in controlled manner.

Types of semaphore.

1. Binary semaphore
2. Counting semaphore.

2. ~~Binary~~ Counting semaphore. varies from $-\infty$ to $+\infty$

* In counting semaphore processes need to runs a program to acquire a resource so there are 2 program

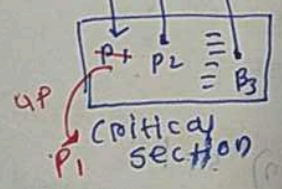
1. P(), Down, wait.

This means if a new process is coming in share region need to run Down program.

Down program

* set a no of value that much process are allow to enter in critical section.

→ P_1 P_2 P_3 $\boxed{P_4\ P_5}$ SR – 3 → 4 → now process goes in CS condition.



P_1 → runs Down program same P_2, P_3

- But when P_4 & P_5 want to use share region it not be allowed to access
- P_4 & P_5 are goes into Suspended state.

CS

* **Up**: 2. V(), up, signal.

program is run when the any process in critical section wants to come out then runs a up program, P_1 comes out. so know the process sends signal to Ready Queue if any process wants to use critical section then used.

1] Binary Semaphore :- 0, 1.

Operations Behave like lock. only take 0 & 1

1. Down

```
{
if (s value == 1)
{
    s.value = 0
}
else
{
    Block the
    process in
    suspended list
    sleep();
}
}
```

2. UP

```
{
if (s.value is == 0)
{   s.value = 1
}
else
{
    select a process from
    suspended list
    and ~~enter to~~
    wake up();
}
}
```

Example $s = \phi$ 1!

P_1 P_2

Down(s) Down(s)

Block.

CS CS

UP(s) UP(s)

queue suspended when $P_2 \Rightarrow s = 0$.

$s = 0$

CS

1st (down) $s = 1$

P2 come

-: Deadlocks :-

- A deadlock occues when ~~each~~ two process wants to get a resouce but not getting.
- This create a dependencies, where no one can get a resouce.

There are four condition to occure deadlock.

1. Mutual Enclusion :- [on recouse hold mutal exculion there]
2. No preemption :- [No prevention there]
3. Hold & wait :- a process holding one resource is waiting to acquire oothee resource.
4. Circular wait

Methods to handle deadlocks :-

i] Deadlock is ignored (when ever occure we ignore for performance (speed)).

ii] Deaadlock peiventions.

- Mutual Encution.
- No peimetive
- Hold & wait
- Circular wait

} try to remove one condition from these or ~~at~~ all condition.

iii] Checking Dedlock avoidance [Bankers Algo]

- Before deadlock occues we check all conditions that create deadlock.

iv] Check first Dedlock detect then allocate.

Close / Unlink Semaphore.

Once the semphore is no longer needed. it should be properly closed and unlinked to free system resources.

```
int sem_close(sem_t *sem);
```

closes the semphore descriptor but it keeps it in system. For than we need to unlink.

Unlink (sem_unlink).

- Removes the named semaphor from the system
- After unliking the samaphore the sem_open( ) is no longer available.

```
int sem_unlink(const char *name);
```

—— x o x ——

Semaphore Applications:- in:

1. Mutal Enclusion (Mutex):
   - for mutex operation to ensure thread are access process at time.
2. Consumer and Producer:
3. Read-write Problem.
4. Resource Allocation: helps to managed access limited number of Resourcess.
5. Event Syncheonization: syncheonised events between threads/ process.

Advantage:

i] Efficient Sycheonization.
ii] No Busy Waiting.
iii] Resource Management.
iv] Deadlocks Avoidance.
v]

Real-World Example:

i] if resourc already lockedor value = 0, the semaphore block the new resource request.

ii] when a recources freed (sem_post) unlink. it signal the sema system that the room is available again.

Threads

- Posix Thread Library (pthread):
  - for managing threads.
  - while processes are independed execution unit threads exists in within process and share same resources (memory and fd).
- Key Characteristics.
  - ~~When a program starts, it begins with main thread~~

1. Shared Resources:
   - Threads in process share:
     - Memory space: All threads access the same globle variables and heap memory.
     - file descripter: if one thread open a file athen other thread can open it.
   - share thread resource make inter thread communication faster but iteroceded challenges race condition and synconitizion issue
2. Main thread:
   - New thread can be created by new thread.
3. Inter - Thread Communication:
   - Threads can communicate in same process easily and efficient than process.
   - Threads can share memory, synconization Mechanisms like mutex and semaphore are required to prevent issues.

Impoetant Note:

- The pthead libeey must be linked dueing a compilation "-lpthead"
- The "-lpthead" and "pheead" Flag ensuee the peogeam can use pthead functions.

Flow:-

1. Ceeatea theead.
↓
Teeds woek
↓
IPC / IPC-Theeds
↓
synchronization - foeee peevention
↓
Theead cycle :- punning, waiting, blocked oe teenina ted.
↓
Linking when compile.

—— xox ——

Theeads Atteibutes in Pthreads:

- Theead Atteibutes aee defince peopeeties that conteoal theead behavioe
- such as stack size, scheduling policy, oe detached state.
- These atteibute as set before creating theeads
- Two common Function to manage theeads:

1. pthead_atte_init( )

Puepose: Initialize a "pthead_atte_t" objt, which will be used to set atteibute of a theead before it created.

Syntax: int pthead_atte_init(pthead_atte_t *atte);

Parameteee: "atte" The pointee to ~~theead~~ pthead_att_t object be initilize.

Return value:

- 0 for success
- perror for : Error code on failuee.

Use: After initilize, calling we set specific, atteibute like stack size before using object in pthead_create( )

Example:

- ~~After initilizing~~ pthread_attr_t attr;
  pthread_att_init(&att);

After init we modify it
1. pthread_attr_setstacksize()
2. pthread_attr_setdetachedate()

2. pthread_attr_destroy( );

purpose: Destroy pthred_attr_~~destroy~~_t object.
- This is important to avoid memory leaks, espessily attribute object no longer needed

Syntax: int pthread_attr_destroy(pthread_attr_t *attr);

Parameter: attr: Pointer to thread attribute object to destroyed.

Return value:
- 0 on success.
- Error code on failure.

Use case: Once thread create and no longer to need to modify its attributes, destroy attribute using these function.

Joinable & Detachable Threads..

- when we creating a thread it can either;
  i] joinable (default)
  ii] Detachable

Joinable Threads

Default type:- Threat are created default joinble unless it detached

Syncheonization:- We use syncheonization the parent thread uses pthread_join() to:
  i] its wait for worker thread to complet.
  ii] Retrive worker thread's exit status.

And the resource managedment is utilize when the worker thread resource is are not automatrally released after finised. it resource are freed when pthread_join is called.

key points:-
i] if we dont called pthread_join then resources leaks

Detachable Threads.

Purpose:- To avoid to having managed thread lifecycle explicitly in the parent thread.

means its not taking care about thread.

No syncheonization:
- The parent thread does not/can not:
  - wait for its completion.
  - Can not retrive its exit status or return value.
- Automatically the resource will be cleanup.

key point:
i] Detaching thread makes its fully indepedend of parent thread.
ii] It used when parent thread has no interest in threads result or completion status.

Life Cycle Diagram:-

i) joinable thread Lifecycle:-

pthread_creat() → Thread Execute → pthread_join() → Resource cleanup.

ii) Detachable threads:-

pthread_creat() → pthread_Detached() → Thread Execut → Resource cleanup.

When to use

i) joinable:-

- When result of thread needed.
- ~~Task~~ Task require synchronization with the parent thread.

ii) Detachable threads:

- Fire-and-Forget task like Logging, I/O tasks.
- No need for the parent to track threads execution or result.

Creating a Detachable threads:

1. pthread_attr_t object.
   - The pthread_attr_t object is used to specify threads attributes, such as stack size, scheduling policy
   - Wether thread is joinable or detached.
2. Detach state.
   - Detach state: The thread resource automatically clear or cleanuped when it terminated.
3. Function Related to Detach state.
   - pthread_attr_setdetachstate().
     - Set detach state of thread.
   - pthread_attr_getdetachstate().
     - Retrives the current detach state of thread
   - pthread_detach():
     - Used to change ajonable to detachbal thread after created.
4. Detached States:
   - PTHREAD_CREATE_DETACHED: When a thread created in a detached state, its resources are freed after the thread executing.

—— xo x ——

i] First, Initilized pthread_attr_t obj

ii] set detached state to PTHREAD_CREAT_DETACHED
pthread_att_setdetached().

iii] create pthread using pthread_create(), passing obj

iv] pthread_detach().

-: Syncheonization Mechanisms :-

Theead Syncheonization Mechanism:

- In multitheead system. The theeads inteeact safely without causing race condition of data curruption.
- Syneeonization Mechanisms use to manage concurrent access to shaeed eesoueee as follow :

1. Mytexes (Mytual Enclusion).
2. Condition Vaeiable.
3. Read/Weite Locks
4. Baeeiens.

1. Mutex/ Mutual Enclusive):-

- A mutex is syncheonization peimitive that peovide enclusive access to shaeed nsoueces in multitheead peogeam.
- It ensuee that only one theead at a time can access the ceitical section (eesoueee shaeed).

Key Fateees of Mutex:-

1. Enclusive lock.
2. Conteolled access
3. Seeial Access.
4. Lock & Unlock Mechnism.

POSIX Mutex APIs:

1. Mutex API

- A mutex was ensure serialised acced to a shared resour by allowing only one thread to hold the lock at a time.
- other thread must wait for the mutex to be unlocked before they acquire.

Operations:-

1. Create :
   - Use static

     pthread_mutex_t mutex= PTHRAD_MUTEX_INITIALIZER;
   - use Dynamicly

     pthread_mutex_init (&mutex, NULL);

2. Lock / Unlock.
   - Locking

     int pthread_mutex_lock(pthread_mutex_t *mutex);
     - Blocks the thread until is available.
     - Return 0 success or error code
   - Unlocking

     int pthread_mutex_unlock(pthread_mutex_t *mutex);
     - Release the mutex, allow other threads to acquire it.
     - Return 0 success

       error otherwise.

3. Destroy the Mutex:

   - cleans up resources when the mutex is no longer needed.

   int pthread_mutex_destroy(&lock);

CH : Signals, Pipes, FIFOs, and mqueue

Signals : signals are software interrupt which sent to a program to indicate that some important event has been occured.

- Some signal indicates that a user has ask the program to do something that it not usual flow of control.
- Depended on source and reason event being signaled.

all signal follow same patteern:-

- The occurance of particular event generate a signal.
- the signal, is delived to process.
- The signal must be handle once it delived.

The interrupt is trigger :-

- User actions (eg.<ctrl+c>).
- system level events(eg. illegal memory access or division by zero).
- Time expiration or inter-process -commun.

Syncheonus & Asyncheonus signals:-

Syncheonus :

- Generated by action within a program, such as illegal memory access.
- Delived to process that caused event.
- Example : Accessing Illigle memory Access.

Asyncheong signal:-

- Trigged by external events unrelated to a programs current execution state.
- Delived to process from outsources, keyboard & timer
- Example: pressing <ctrl+c> to interrup a program
- This signal arrives unexpectedly and can not be predicted by reciving process.

How signal are Handles :-

1. Generated: An event generate a signal.
2. Delived : The signal sent to process
3. Handled : The process responds to signal.

signal Handlers:

i] Default signal Hander

- Every signal has a default action predefined by system kernel.
- Example: A segmentation Fault (SIGSEGV) terminat program by default.

ii] User Defined Handler :-

- A Developer can override the defult action by specifying custom behavier for signal.

Example:- A program could save its state before termination. when Recive termination (SIGTERM)

Who can send signal

- kernal (Default signal Hander)
- Process with sufficinet permission

Signals on Linux :-

- Signals on linux are mechanism for communication between the kernal & process or between process.
- That represent various event (signal) that need to be handled

Total No of signals:-

1. Linux defines 64 signals.
   - These signals are declared in <signal.h> header file.

Signal Identification:-

- Each signal is identifed by unique integer value range from 1 to 64.
- Signal also have describe names with prefix SIG (eg. SIGINT, SIGKIL).

Signal Categories:-

- based on the range and purpose this signals are divided.

1. Standard signals (1 to 31):
   - Used by kernel for pre-defined events.
   - Example:
     SIGKILL: Forces a process to terminate.
     SIGSEGV: Segmentation fault.
2. User predefined signals (32 to 64):
   - Reserved for user-defined purpose or inter-process communication.
   - These are real-time signal, allow process to send specific info to other process.

Kernel generated signals:-

When specific event occurs between rang (1-31) the kernel sent the signal to the process.

Example of Kernel Generated signals:-

1. SIGSEGV : segmentation fault
   generates /Indicate invalid memory access by a process also know as segmentation fault.
2. SIGINT: Indicates terminal interrupt, sent when user press <ctrl + c>. running process get stop.
3. SIGCHILD: when a child process terminate or stop sent to parent.

Terminology In Linux:

It explain how signal is generated, managed, Handled.

Signal Generation:

- A signal is generated when specific event occurs.

Signal Delivery:

- After generated, a signal is delivered to the target process. it take response may terminate or execute by Handler decided.

Pending signals:-

If it generated but has not delivered to the press.

Blocking signals:-

- A process can block specific signals by adding them into signal mask.
- it can not delivered to process.

Signal Mask:

- A signal mask is a set of signals that process has choose to block temporarily.

it specifies by process using system calls

- sigprocessmask( )
- pthread_sigmask( )

Unblocking signal:-

- Once a signal is removed from signal mask any pending signals are imediately delived to process.

——— //x// ———

Signal Actions (Disposition):-

- The behavior or action taken by process upon recieving a signal is called signal disposition.

The possible Action include:-

1. Ignore the signal

The process ignore the signal and continues the execution without taking specific action.

2. Terminate the process.
   - The process terminate or get killed upon recieving a signal.
3. Core dump or Terminate.
   - The process generate core dump (a file containing the process's memory state). it is then terminated.
   - Core dump are used for debbugging to analyzes the state of process at the time of termination.

   Example: segmentation fault.
4. Stop the process:
   - The process execution is suspended (put into stop state).

   Example: SIGSTOP

5. Continue the process.
If a process in stop state, recieving certain signal causes it to resume execution. SIGCONT

6. Default Actions:
- Each signal has default actions depending on signal.
  - SIGKILL :- Terminate
  - SIGSEGV :- core dump & terminates
  - SIGSTOP : signal stop.

Standard signal Overview:-

SIGPWR → Terminate power failure

Signal Name	No.	Defaut Action	Desceiption.
SIGHUP	1	Terminate	Hangup detect on cont. terminal
SIGINT	2	Terminate	Interup from keyboark ctelt C
SIGQUIT	3	Core Dump + termina	Quit from keyboard ctelt =
SIGILL	4	-||-	Illigal Instruction
SIGTRAP	5	-||-	Trace or breakpoint trap.
SIGABRT	6	-||-	Abort signal from abort() fun.
SIGBUS	7	-||-	BUS error (Invalid Mem Access)
SIGFPE	8	-||-	Floting point execption
SIGKILL	9	Terminate (Not Ignore)	Kill signal. force terminate.
SIGUSR1	10	Terminate	USER DEFINE SIGNAL 1.
SIGSEGV	11	Core dump and ter.	Segmentation fault.
SIGUSR2	12	Terminate	User defined signal 2
SIGPIPE	13	-||-	Write a pipe with no reader
SIGALRM	14	-||-	Timer signal from alarm() fun
SIGTERM	15	-||-	Termination signal
SIGTKFLT	16	-||-	Stack fault on coprocess.
SIGCHLD	17	Ignore	CHILD PROCESS stop.
SIGCONT	18	Continues	Resume stopped process.
SIGSTOP	19	stop (Not Ignore)	stop a process (like pause)
SIGTSTP	20	stop	stoped typed at terminal ctelt Z
SIGTTIN	21	stop	Background process read from termi
SIGTTOU	22	stop	Background process write to termi
SIGURG	23	Ignore	urgent condition on soket.
SIGXCPU	24	core dump & term	CPU time limit exceed
SIGXFSZ	25	-||-	Fix size limit exceeded
SIGVTALRM	26	Terminate	Virtual Alarm
SIGPROF	27	-||-	Profile timer expired
SIGWINCH	28	Ignore	window size change
SIGPOLL	29	Terminate	Pollable event occured
	30	-||-	

Key points:

1. Default actions:-
   - Terminated : End the process
   - Ignore - Discarded
   - Stop :- Pauses the process execution.
   - core dump: Generate a core dump.

2. Non ignored signal.
   - SIGKILL and SIGSTOP. and SIGCOUNT.

Signal Handling:

- signal Handling is a process by which program react to signals sent by default.
- Every signal has predefied action but some signal the deufit action can be overside.
- This involve defining custom (Handler) that will execute when signal delived to process.

So defaut action: Default behavier

Custom signal Handler: A user can defined a funtion and handle a signal differently by default.

This know as "catching".

Default signal Handler function:

```
#include<signal.h>
void signal-handler(int signum){
}.
```

Register signal handler

signal (SIGINT, signal handler).

Changing Actions of signal:

it means modifing the default behavier a process takes when it recive a signal like its default behavier core dump.

So Linux provide mechanism to change these action to handle signal in custom way.

By using API:- sigaction

Example/syntax:

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *new_act,
              struct sigaction *old_act);
```

i] sig: The signal number whose action want to change (eg, SIGINT, SIGTERM).

ii] new_act: A pointer to a structure struct sigaction that specific new action take for signal.
set NULL if no changes are needed.

iii] old_act: A pointer to a struct sigaction that stores previous action.

Structure Definition: struct sigaction

```
struct sigaction {
    void (*sa_handler)(int) //signal handler function
    sigset_t sa_mask; //signals to block during handler execution
    int sa_flags; // Flags to control handler behavier.
};
```

Field Enamplained:

1. sa_handlee:
   - specifies the handlee functions:
   - Can be
     - A use-defined function pointee (eg. my_handlee (int sig).
       - SIG_IGN : Ignoee the signal.
       - SIG_DFL : Reset to default action.
2. sa_mask:
   - specifies signals to blocks while this signal handlee is running.
3. sa_flags:
   - flags affecting the signal handling behaviee.

Ignoeing Signal:-

Ignoeing a signal means telling OS that the peocess does not want to handle that signal discaed it.

This is achived by setting the sa_handlee field of a steuct sigaction to ~~sig~~ SIG_ING.

Signal Desceiption.

Linux peovide human deeadble desceiptions foe signals using the stesignal API.

API foe peinting signal desceiption:

```
#define
#include <string.h>
~~parameter:-~~ ~~sig~~ char sig ~~*sig~~ *stesignal (int sig);
```

Pipes

Overview:

- Pipe is a mechanism for unidirectional interprocess communication (IPC) it allow data flows in one direction between two process that is known as unidirectional IPC.
- One process acts as sender and other is recivER (reader) : (writer)
- within same channel two data exchange.
  - Usage Example:-

    ① ls | wc -l. ②

- The shell create two process fork() and exec() connect them using pipe (ls and wc).

Characteristics of Pipes:-

1. Unidirectional:-
   - Unidirectional data flow one way only.
     - End write: Used by writting process.
     - Read End: End by reading process
2. Data flow continous stream the data we ~~write~~ ~~see~~ read sequencily.
3. Pipe capacity
   kernel maintain fined size buffer for each pipe

The two type of pipes are:-

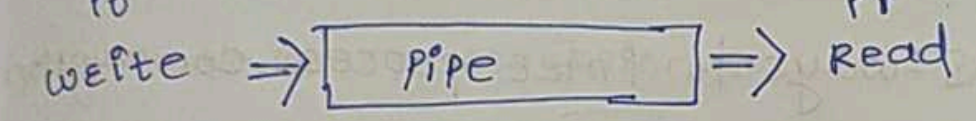
1. Unnamed Pipes:-
. This files do not have backing files and exists where process is running used for communication
1. I C programing pipes are created using pipe() syscalls.

Unnamed file created in shell "|".

ls | grep.

—xox—

Pipes



- unidirectional.
- child process communication.
- for communication. pipe the system call is used [pipe(int fds[2])] fd[0] & fd[1] read write.

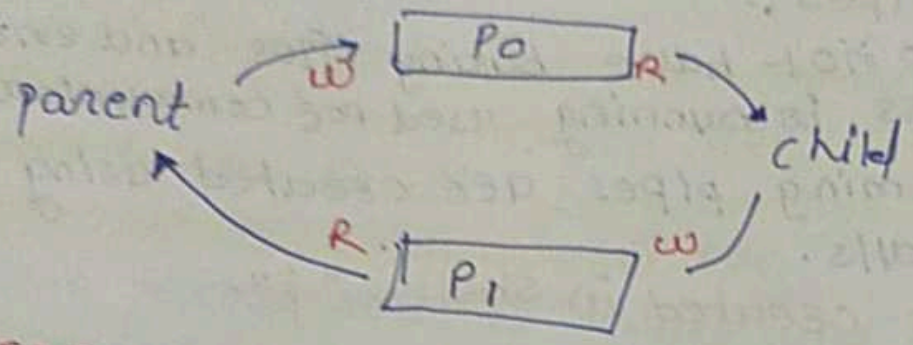
Return "0" success
"-1" failure

- This communication happens by FIFO Principle first come(In) first out.

- System calls used for pipe
  1. write() Fd[1]
  2. read() Fd[0]

Return: It returns how many byte are write/read during system call.

structure of two way communication through pipes()



APIS :

write (), read (), open (), close () APIS are used.

There are 5 ways to inter-process communication

1. Pipes
2. Message Queue
3. Shared Memory
4. Sokets.
5. Semaphores.

:- Message Queue

Synchronous and Asynchronous Transmission.

Syncheonous Transmission: In this data is sent in a block of frames.

- syncheonus transmission is fast.
- syncheonus transmission in this time interval is constant.
- In this users have to wait till the transmission is complete before getting a respance back and there is no gap present between data.

Asyncheous Transmission:-

- Data sent in a form of bytes or characters.
- Slow transmission.
- Time interval is not constant.
- User do not have to wait for completion of transmission in order to get response.

There are one more also we use both syncheonous & Asyncheous both hybride.

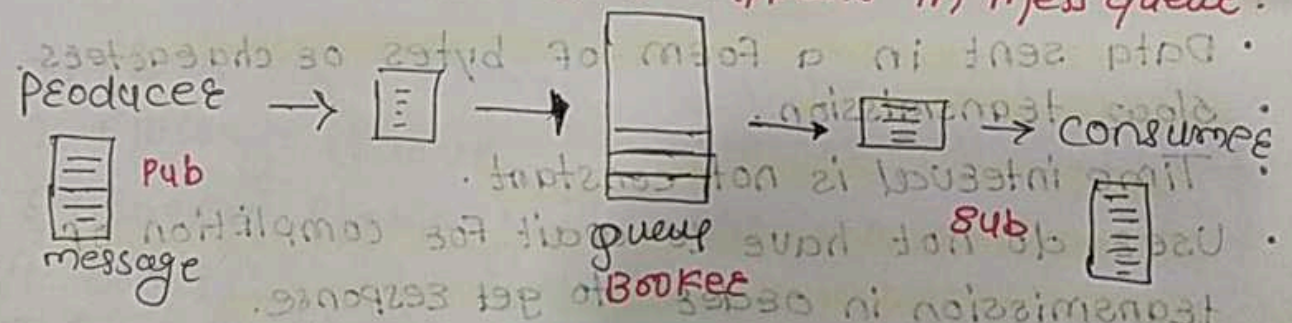
-: Message Queue :-

- A message queue is communication and data transfer mechanism used in a system design.
- Message are sent between various programs, systems inside a software architecture.
- These message contains two parts:-

Header: These contain metadata of message such as unique identifier, timestap, message type and routing info.

Body: The body contain actual data payload. It can be any format including text, binary data.

Asynchronous communication happens in mess queue.



- Sends a message in queue then the message is store temporary making available for consumer when the message consumer receive message from queue.

- In a message queue assigned a priority levels
- when consumer wants to fetch data based on their priority data can be fetch with 0 is low priority.

Posix Message Queue in Linux:-

- Posix message queue provides a mechanism for inter-process communication that allow message process to exchange data in the form of message.
- message queue support priority prioritization and donot enforce strict FIFO

Message Queue Functions:-

1. Create/open message queue:-

```
#include <fcntl.h>   // For O_constants
#include <sys/stat>  // for mode constant
#include <mqueue.h>
mqd_t my_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

- name: Name of message queue. "must start with /"
- oflag: flags for queue operations ("O_CREAT to create a new queue).
- mode: Permissions (similar to file permission)
- attr: pointer to structure of attribute.

Example:-

```
struct mq_attr attr = {
    .mq_flags = 0;      // Blocking mode
    .mq_maxmsg = 10;    // Max no of mesg
    .mq_msgsize = 256,  // size
    .mq_curmsgs = 0     // No of message currently in queue.
};
```

2. Send a Message Queue:-

```
int mq_send(mqd_t mqdes, const char msg_ptr[], size_t msgler
            unsigned int msg_prio);
```

mqdes: message queue descriptor.
msg_ptr: pointer to the message.
msg_len: length of the message.
msg_prio: priority of message.

Example:-

```
char *msg = "Hello"
mq_send(mqd, msg, strlen(msg), 5);
```

3. Receive a mesq from queue.

s mq_recive(mqd_t mqdes, char msq ptr[],
size_t msq_len, unsigntint *msq_prio

where
msg_ptr : Buffer to store recived message

Example:-

```
char buffer[256];
unsigned int prio;
mq_receive (mqd, buffer, 256, &prio);
Pf ("Recived message : %s with prio", buffer, prio);
```

4. close message queue.

int mq_close(mqd_t mqdes);

5. Unlink (delete) a message queue

int mq_unlink (const char *name);

Socket

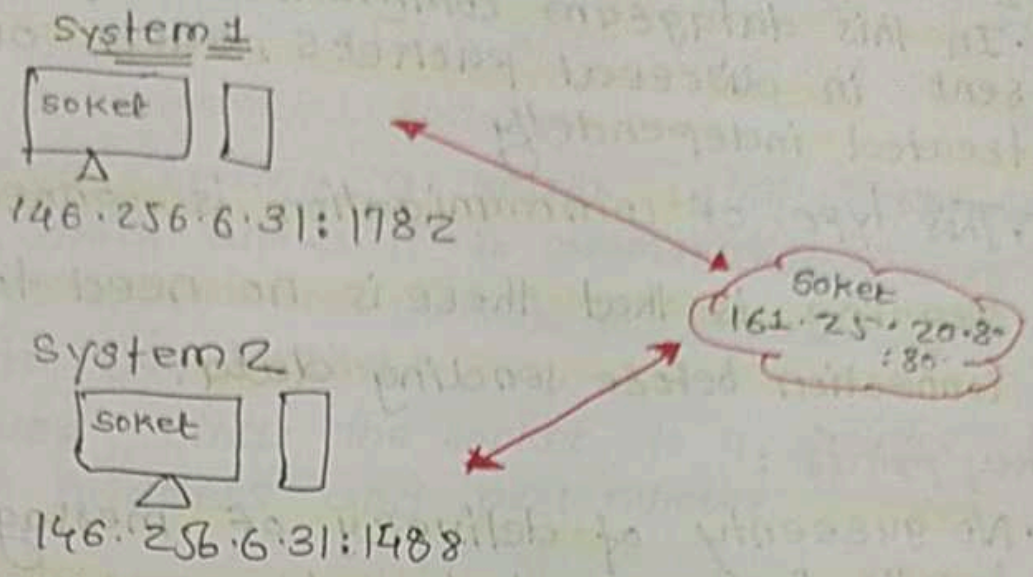
what is soket :-

- A soket is vietual endpoint foe sending oe reciving data aceoss netwoek.
- It a peogeamming inteeface that allow communitie between two systems oe peocess either on same machine oe across a different machines connected by a netwoek.

—//—

- A software structuee (soket) within a N/W node of a computee netwoek that seeve as a endpoint between sending and reciving data across the netwoek.
- A soket is exteenaly identify to othee hosts by socket addeess.
- Socket Addeess is the combination of an ip addeess and a poet numbee.
- Combining a teanspoet layee poet numbee and the netwoek IP Addeess uniquly identify paeticulae application running on an individual host device. This combination called socket.
- The bidieectional communication happen using a socket ovee a netwoek.

Communication Architecture of Soket



- Each system have there unique ip and port no assignet to socket. they communicate over a network.
- so if we think system 1 communicate with system2 then they connected over a n/w.
- Based on these port the message delived properly over a n/w.

—//—

~~Soke~~ Socket N/W communication types:

1. Steeam communication:

In steeam communication data is sent as a contious flow of bytes. This type of connection is a connection oeiented before data transmist the connection must be establised.

key points:-

- Data is gurunteed to arrive in currect order and without duplicating.
- TCP primary protocol is used for steeam.

2. Datagram Communication.

- In this datagram communication data is sent in discreat packets, each of which treated independelly
- This type of communication is connectionless
- Meaning is that there is no need to establish connection before sending data.

Key points :.

- No gureenty of delivery of message (data). but its fastee and has lowee overhead.
- This is sutable for application where speed is more impoetant reliability.
- UDP is data protocol used for datagram communication.

3. Raw Communication :-

- Raw soket allow direct sending and reciving of IP ~~address~~ packets without any protocol specific transport layer formating
- This type of communication used for low level N/W communition.

Key points:-

- Works directly on IP packets.
- Complete control sent and reciving data.

steps to ~~d~~ setup client communication with socket: - [server side - Connection N/W on
client side - Listening.]

The server side:-

1. Creates a Networks soket:
   - The server creates a soket using a programming function, which allow it to communication over network.

2. Blinds to this soket:-
   - The server binds the socket to a specific IP Address and port number.
   - The IP address identifies the machine on the Network and port number specifies a particular service or application on that machine.
   - above 1024 are often used for custom application ports below 1024 are reserved for standard services.

3. Listens on this soket:
   - The server soket is put into listening mode,
   - It can handles: multiple connection clients trying to connection.

4. Accept connection from client.
   - when a client tries to connect, the server accept the connection, establishing comm link.

The client side:-

1. Creating N/W soket:-
   The client soket also create for comm with the server.

2. Connection to the server:-
   The client soket tries to establish a connection with the server using IP and port number

1. Socket Creation: socket( )

Soket programming :-

1. Soket Creation : socket( );

Function prototype :-

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

Purpose :

This function create an soket for endpoint communication.

Parameters :

- Domain: Specifies the communication domain. refers to communication protocol or enviement in which the socket Operate. It defines the set of rules or addressing scheme that socket will use to communicate.

  comon socket domain:

  - AF_UNIX or AF_local : For IPC on same machi
  - AF_INET : For interet IPv4 protocl
  - AF_INE6 : For interet IPV6 protocol.
  - Other domain like:
    - AF_CAN: Communication N/W. over can
    - AF_BLE : Blothooth communication.

- type: Specifies type of socket.
  - SOCK_STREM: For TCP (reliable).
  - SOCK_DGRAM: For UDP (unreliable)

- Protocol: specified protocol to used.

  0 : Automatic select the appropeite protocol

  IPPROTO_TCP: For TCP

  IPPROTO_UDP: For UDP.

- return: FD file descriptor for success

  errno "-1" for failure.

2. Bind the Socket : bind ( )

- The "bind ( )" Function is used for associate a socket with specific local address and port number.
- This is crucial step for servers because it tells O.S that the socket should listen for incomming connection on port.

Fuction prototype:

```
int bind(int sockfd, const struct sockaddr *addr,
                socket_len addrlen);
```

Purpose: Assigned a specific address and portno.

Parameters:

- sockfd : socket file descriptor returned by socket( ).
- addr : A pointer to socket addr contain a address and port.
  for ipv4 struct_sockaddr_in
- addrlen : The size of addr structure.

```
struct sockaddr_in {
    sa_family sin_family; // Address family
    in_port_t sin_port; // port no.
    struct in_add sin_add; // IP address
}.
```

Return: "0" on success
-1 on error.

3. Listen (): Listen for connections : Listen();

- listen() function is used prepare a server socket to accept incoming client connection.
- Instead of sending data it will be create a connection.

Function prototype:-

int listen(int sockfd, int backlog);

Purpose:

Accept incoming connection.

Parameter

sockfd : The socket file discriptor.
backlog : The max no of connection.

Return:-

0 for success.
-1 for errno.

4. Accept a connection accpt();

- accept() fun is used by server to accept an incoming client connection on listening socket.

Function prototype:

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

Purpose: Return new socket descriptor.

- accpt() commcition with client.

Parameter:-

- sockfd :- File descriptor
- add : A pointer to sockadd structure. client address store where
- addrlen :- Size of addr structure.

5. Convert Host to Network Byte Order

- In network programming function like htonl() and htons() are used to convert values from host byte order to network byte order.
- This is important because diff compute Arch may use different bytes order (endianess)
- And n/w ~~bytes~~ protocol requires a consistant byte order (big-endian) to ensure that data is transmited correctly between systems.

Function prototype:-

```
uint_32_t  htonl(uint32_t hostlong);  //convert a long Integer.
uint32_t   htons(uint16_t hostshort); //convert to short Integer.
```

Example:-

```
uint 16_t  port = htons(8080);
```

6. Connect to server: (connect());

Function prototype:

```
int connect(int sockfd, const struct sockaddr *addr
            socklen_t addlen);
```

Purpose:-

connect() is used by the client to establish a connection to a server.

Parameter:-

- sockfd : The socket file descriptor.
  addr : A pointer to server address structure
  addrlen : The size of address structure.

Return value:-

"0" on success
"-1" errno.

7. Convert Hostname to IP address

The hostbyname() function in C to convert a hostname (www.example.com) to there corresponding ip address.

Function prototype:-

```
struct hostent *gethostbyname (const char *name);
```

Purpose:-

- gethostbyname() hostname to IP address.

Return value:-

- Return a pointer to hostent
- NULL on error.

Structure of hostname:-

```
struct hostent {
        char *h_name; // offical name of the
        char **h_aliases; // host
        int h_addr_type; // Alise list
        int h_length; // Address type
        char **h_addr_list; // length of address.
}                           // List of address.
```

Scheduling & Scheduling Polices

Scheduling:- Schedulling in O.S refers to by which processes are given access to system resources, primely CPU

- The O.S uses a scheduling to decide which process runs at a given time.
- In simple word its manages the process execut from memory (RAM) to CPU. The process are wait in ready queue to the schedulling decides to schedule a process.

Impotance of Schedulling in O.S:-

- Resource Management:-
  - Schedulling ensure allcation of CPU utilization allocation to various process.
- Responsiveness:-
  - Good scheduling can improve the responcetime of system.
- Throughput:-
  - By minimizing ideal CPU time and maximum CPU utilizat increase throughput:.

Throughput:- No of process that are completed or executed by CPU in per unit time.

- Fairness refer to equitable distribution of CPU time and other system resources among all process.

  ensure that ~~CPU~~ processes get a fair share of CPU time
- Helps to manage process based on there priority.

The Role of Schedullee:-

- Selecting a next process to run based on these algo.
- managing process states, such as running, ready and wait.
- Context switching: Schedullee Handles the saving of one process and loading another process during context switch. means its shift from executing process to another.

—x o x—

CPU - Schedulling Basic

Types of Schedulling:

1. Preemptive Schedulling:

- Define: Preemptive schedulling is like when a CPU executing a process and suddenly suspend the process or stop the execution of process these know as preemptive schedulling.
- Scenario: when a "A" process is running in cpu or executing during these any sudden process are come like interrupt then that time CPU stops the execution and executing from these. a new process and when a process "B" complete then a "A" runs on CPU.

Algorithms:-

✓ 1. Round Robin

✓ 2. Short Remaining Time First (SRTF).

3. Largest remaning Time First.

✓ 4. Priority Based.

Non-preemptive Schedulling:-

- In a non-preemptive schedulling a process are in a CPU executing. in between the process execution no other process are running.
- Once the process is start executing it will run to complection.

Algorithms :-

1. First come first serve (FCFS).
2. Shortest Job ~~next~~ first (SJF)
3. (LIF) Largest job first
4. HRRN : (Highest Responce Ratio Next).
5. Multilevel Queue.

—//— //—

Key schedulling time/criteria:-

1. Arrival time:- A time at which a process enters into Ready Queue or stack.
2. Burst time:- Time required by a process to get execute on CPU. "Duration of execution"
3. Complition time:- A process get completed its execution.
4. Turn-around time:- The total time taken by process to complete from Arrival to complition time.

[Turn-around Time = Complition Time − Arrival time]

5. Waiting Time:

The total time a process spends in a ready queue before it gets CPU execution time.

[Waiting Time = Turnaround Time − Burst Time]

6. Responce Time:

- Its a time at which a process get CPU first time from Arrival time.
- Responce Time = [Time at which process get CPU time − Arrival time]

7. CPU Utilization:

The percentage of time the CPU is actively working on processes.

8. Throughput:

The no of process completed per unit time.

Schedulling Algorithms:-

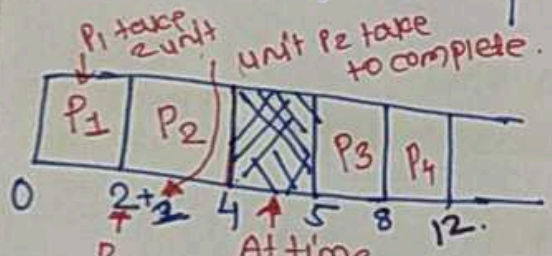
- Algorithms are used to schedulle the processes based on Algo, priority, and i/ps.

1. First Come First Serve:-

- The process arrive first is schedule first and Runs until it finish. its a non-preemitive schedulling
- When first process is scheduled that kept in a queue and the process at front of queue gets CPU first. Once it enter in CPU it will run until gets Completed.

Example:-

Process No	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
P1	0	2	2	2	0	0
P2	1	2	4	3	1	1
P3	5	3	8	3	0	0
P4	6	4	12	6	2	2



P_1 taking 2 unit to execute in CPU

At time unit of time not any process is arrive so it idealy

Impoetant

- Time is running in 1 unit at perticular unit the time. the arrival time process get scedulled.
- In between completion time no processes is scedudle so process goes in idealstate not execute any instruction

- In between "P_1" that time 2 unit are take so the next Pococess that between 2 unit time was schedued in ready queue.

2. Shortest - Job First : (SJF) :-

Define : SJF scheduling select the process with shortestet burst time for execution next.

Type : Preemptive and Non-preeptive.

Example :-

Process No.	Arrival Time	Burst Time	Comp Time	TAT	WT	RT
~~P1~~	1	3	6	5	2	
~~P2~~	2	4	10	8	4	
~~P3~~	1	2	3	2	0	
P4	4	4	14	10	6	

	P3	P1	P2	P4

0 1 3 6 10 14

- from 0 no process arrived.
- then check which process arrived at 1. two process are arrived that is P_1 an P_3. so know we check the criteria that is Burst Time. at this P_3 get less amount of time so P_3 get scheduled.
- know from this duration 1-3 the check which process get arrive so the P_2 is arrived. then check criteria so burst time of P_1 is less so P_1 take CPU
- know P_2 P_4 Remain both have equal time criteria so know check who is arrive first and schedule that to CPU.

Note in Algo LJF is same as these but in these the Large Burst time for execution.

Round-Robin Algorithm :-

- CPU scheduling algorithm where each process is cyclically assigned a fixed time slot.
- Round Robin algo all process arrive at same time
- The main focus on dividing CPU time equally and manages the process queue.

Characteristics of Round Robin :-

1. Equal Time allocation :- Each process get equal time slice to execute.
2. Cyclic Execution :- process are schedule in circular order, and CPU moves next process in the queue after compliting a time quantum
3. No process starvation :-
   - All processes are guaranteed CPU time at regular interval.
   - Preventing any process from being neglated
4. Start time :- There is no need to consider a arrival time but depend upon some case the arrival time need to be considered.
5. ~~Charac~~ frequently context switching occur from CPU to ready queue.

Example:-

Process No	Arrival Time	Burst Time	Comp Time	TAT	WT	RT
P_1	0	5 ~~3~~	12	12	7	0
P_2	1	4 ~~2~~	11	10	6	1
P_3	2	2 0	6	4	2	2
P_4	3	1 0	9	5	4	4

Ready queue

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~	P_4	P_2	P_1

0

CPU running queue:

P_1	P_2	P_3	P_1	P_4	P_2	P_1

0 2 4 6 8 9 11 12

t →

- during P_1 – 0 – 2 – Tq time 2 process are arrive that is P_2 and P_3
- know context switching happen stop the P_1 process in goes to PCB ~~and~~ these info save.

* The main Think in this is Time quantum every given time quatum CPU gets switch to process

if we have

Tq = 2 unit

in every 3 unit CPU swith.

—||x0x||—

Look at

we take arrival 0 process from ready queue to CPU. it only execute upto Tq = 2 unit then it during 2 unit

if any process occur that will be take next process execte upto 2 units.

—||—

after every Tq check that time any process is arrive if arrived put them in read queue but before that if the running process time remaining for execution put I in ready queue then new one.

Scheduler Goals :-

Depending on system requirement a scheduler in a OS achieve various goals

1. Maximumizing Throughput:-

The objective is to maximize the amount of work completed by per unit time.

2. Minimizing wait time:-

This goal focuses on reducing a time a process waits before it starts execution.

3. Minimizing latency.

reducing the responce time of the system.

4. Maximizing Fairness:-

~~For All~~ Ensure all process get a fair share of the CPU time.

5. Miting Deadline -In Real Time enviroment.

- In a real-time enviroment.
- deadline is crucial to ensure.

—— x o x ——

Scheduling in Linux:-

- manages by queue of threads
- scheduling based on thread, not process.
- Each thread have schedulling policys.
  - Real Time & time share.
- R-T Task have high priority & preemission ~~allow~~ allows to switch High priority Task.

Time sharing Vs Real-Time.

Time-Share:

- focuse on faireness, ensuring each thread get a faire share of CPU time.

Complete Faire Schedullee (CFS):

focus on foireness.

SCHED-NORMA :- Default policy

SCHED-BATCH :- batch jobs for large time guaranty.

SCHED-IDLE :- Low periority.

Real-Time

- Based on periority, ensure deterministic behaviee.
- Polices include

1. SCHED_FIFO : first in first out.
2. SCHED-RR : Round Robin.

Niceness is value in linux that determine the periority of process in content of time shared scheduling.

The niceness range from -20 (most nice) 19 (Least nice).

-20 : more CPU time.

19 : less CPU time.

Memory (RAM) Management

The role of memory management is ensure that application have enough memory to execute. to alloca memory as needed. and deallocated when no longer needed.

- The main purpose of memory management is that as no of possible process are kept in RAM/memory for better ~~cast~~ CPU Utilization.

The degree of multiprogramming:-

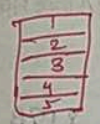
Its refer as number of process kept in a memory simultaniously. A Higher degree of multiprog. generally leads to better CPU Utilization.

Example:-

- Suppose a one process Running and its contentain some i/o operation. so for that its take time so for that operation time to execute our CPU get ideal state (no process execute)
- so to resolve these we kept no of process in a ram that each process simulatiously get execute. as no of process in memory its easy to achive better throughput.

Memory Management Techniques

- Continous
  - Static
    - fixed partition
  - Dynamic
    - Variable partition



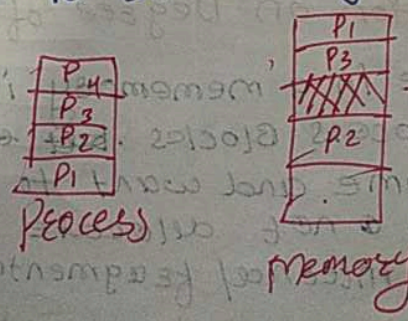
- Non-Continous
  - Paging
  - Multilevel paging
  - Invert paging
  - Segamentation
  - Segmented paging

1] fixed partition: fixed size of BLOCK allocate to process.

2] Variable partition: variable size we allocate during runtime

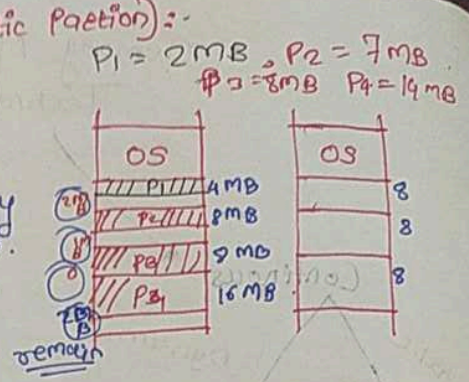
Non-Continuos:-

Non-continuos allocation a process is not continously store in memory it will divide in subparts and the subparts are dynamically allocate to in memory.



1] Fixed partitioning :- (static Partion) :-

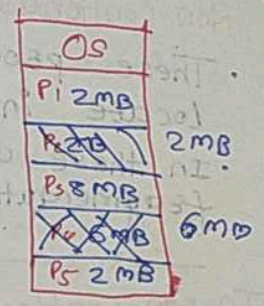
- No of partition are fixed.
- pre-configured partition
- No of partion size is may or may not be same.
- Contious allocation of space is not allowed



—x o x—

- In a memory suppose the P_1 process have size of 2MB so in a memory, We take space of two 2 MB but the remaining size of that block are not used this called as a internal fragmentation.
- P_2 = 7MB are come and accupy the space 8MB block in these also internal Fragmentaion occurs.
- If we Process "P_3" = 32 MB but in our memory there is no space or block available for 32 MB size so there occur a Limit in process size.
- There is a limitation on a Degree of programing because in fixed partioning if we have only 4 blocks then the process only 4 occupy but greater than these not occupy it leads to Limitation on Degree of programing
- After having a space in a mememory in a chunks in a process Blocks. of 5MB size. If 5 process come and want to take space in memory but not allocate a space because of internal fragmentation so there occurs a External fragmentation case.

Dynamic partitioning or variable partitioning



- In dynamic partitioning we allocate size to the process on Run-time
- when the process are running that time we dynamically allocate ~~spc~~ size.
- According to need of process size the size allocate on run-time.

—xox—

- There is no internal fragmentation.
- There is no limitation of processes.
- There is no limitation of process size.

—xox—

- If we put 5 process in memory they have these size allocated. But suddenly P2 & P4 process get completed and deallocate a resource. know P6 process come with size 8MB ~~but~~ we have space in memory but we can not allocate beause of contionus memory allocation. thise issue know as External fragmentation.

—xo—

External fragmentation occues due to memory does not have contiouse memory allocation.

Holes are free space in memory.

Non-Contionus Memoey Allocation:-

- These peocess are divided into paet are locate in different location in a memoey
- In these we tey to remove external fragmentation.

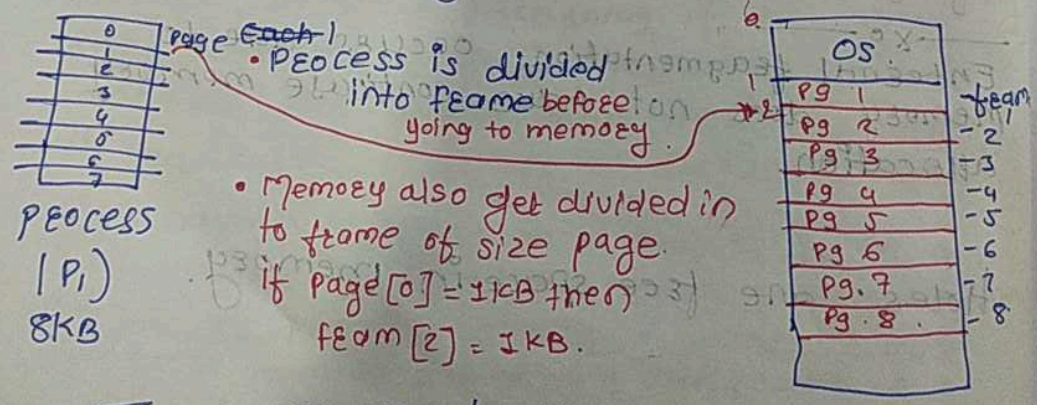
—xox—

In non-contious memoey Allocation we use a mechanism called Pagging:

—xo—

Paging:-

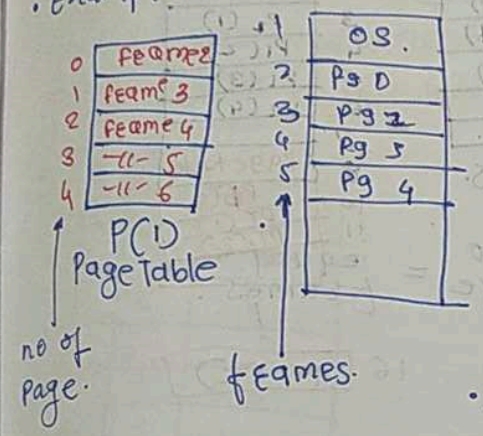
- The peocess are divided into paets each paet know as a page before going to in a main memoey.
- In main memoey is divided in to a frames
- No of frame size is equal to size of page should be occur.
- Suppose we have peocess "P_1" size of 8KB and have a memoey



- The peocess are load on memoey.

Page Table:-

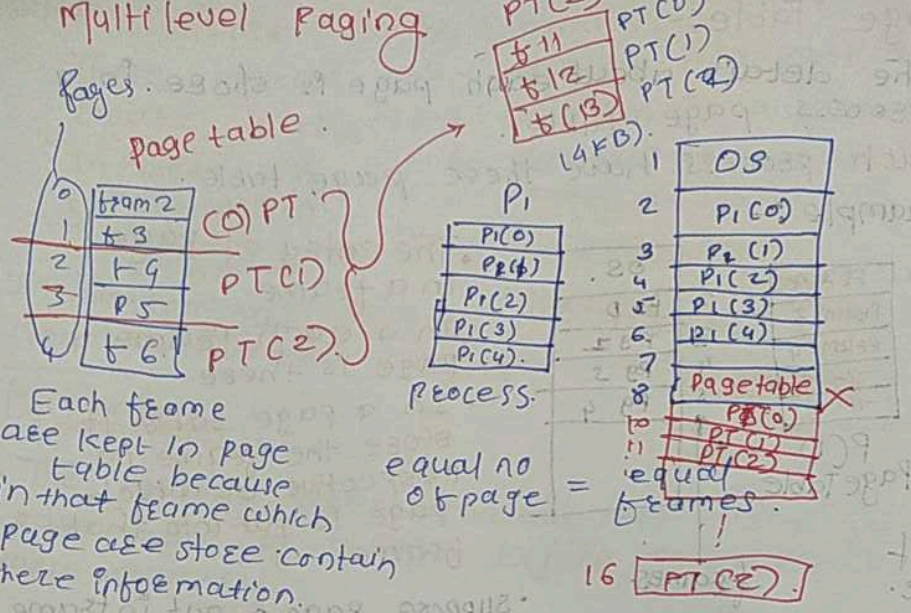
- The detail about each page is store in a process page Table.
- Each process have there page table.
- Example :-



- The entry of page in a frame.
- In a each frame a page is there.
- In a page table it store the frame no respective of which page is put into that frame.
- Suppose page 0 put in frame no 2. so page 0 detail store in frame 2 and frame 2 address store in page table.

- The logical address converted into a physical address of frame and store into a page table.
- It help to find out the pages in memory.

Multilevel Paging



Each frame are kept in page table because in that frame which page are store contain there information.

equal no of page = equal frames.

The problem is here our page size = 4KB and our page table size contain 12KB.

we dont put 12KB size table in 4KB frame so for that we need help of multilevel paging.

Know we divided the page table into no of page table parts. & That page table parts entry kept in a other Pg Table and that frame contain in a

PT base Register

Inverted Page Table

- In this page table a globle page table are declared and used for the paging.
- In page table are in ram memory contain various frame location so its hard to find out frome there. So this inverted page table are used.
- These

frame No (index)	Process ID No
0	P_0	P_1
1	P_1	P_2
2	P_2	P_1
3	P_1	P_3
4	-	
5		
6		

~~No of~~ No of frame = No of Inverted Entry

$\boxed{120 = 120}$

Page Demand

In a memory various pages are there and during some execution time it needed a page so that not present in memory occurs page fault.

Page Demand: Depending on a requirement of a page in MM for execution before that time the process in take from Second Memory

Particular page is untile not required dont put on MM that know as Demand paging.

If degree of program is running

→

Pg1	P1
Pg2	P1
Pg3	P1
Pg4	P1
⋮	⋮

memory

- When a in a memory have process & three page no but sudden CPU want pg no 2 in process no 1 this occure page fault

THRASHING:-

Occurs when the multiple page fault occur and O.S is busy to serve pages from memory to main memory this know as THRASHING.

(second eg RAM)

—//—

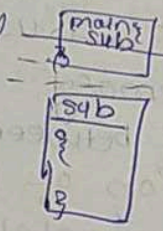
How we remove thrashing:-

- ~~Dur~~ Increase MM space.
- Long Term Scheduler
  - Control करो Degree of Programming.

Segmentation:-

In paging suppose we have program of Add

```
main
{
  sub( )
```

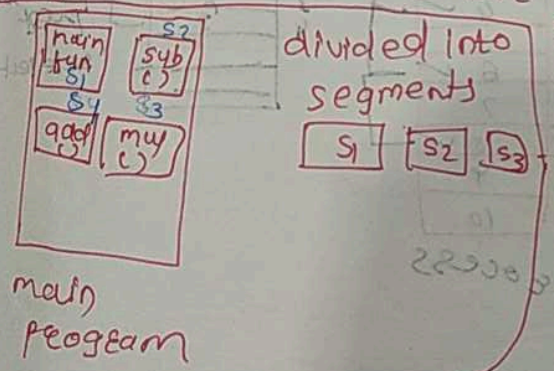


we divided the sub( ) fun two parts in page. in pg(1) have function in pg(2) have operation this leads a page fault.

So for that we use segmentation.

segmentation:-

we divide the memory in to different segments. these segment logical division such as array, functions or data structure.

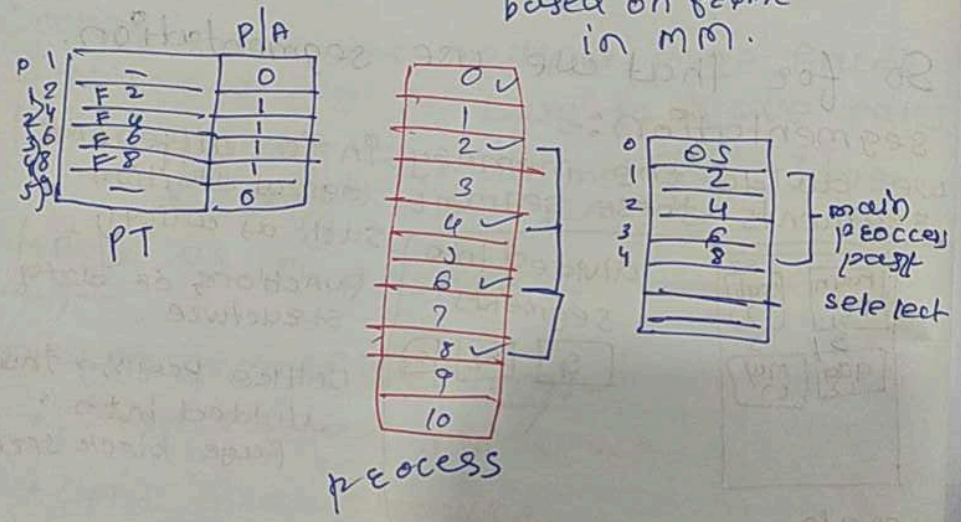


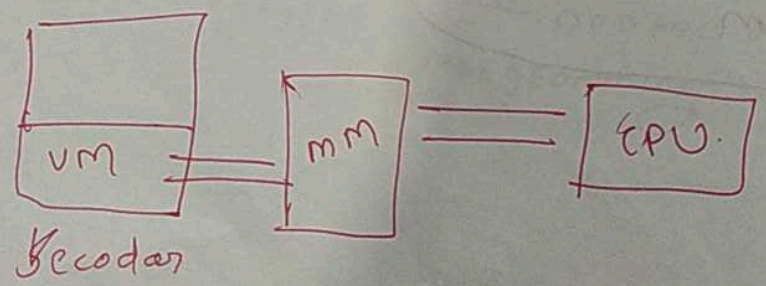
Unlike paging that divided into page block size.

Virtual memory :-

- It is a memory management technique that create illusion for a user of a very large (main) memory.
- Concept is that pages divided between RAM and disk.
- This allow the execution of processes that may not be completely in main memory.

Present and Absent based on frame available in MM.





Benifits

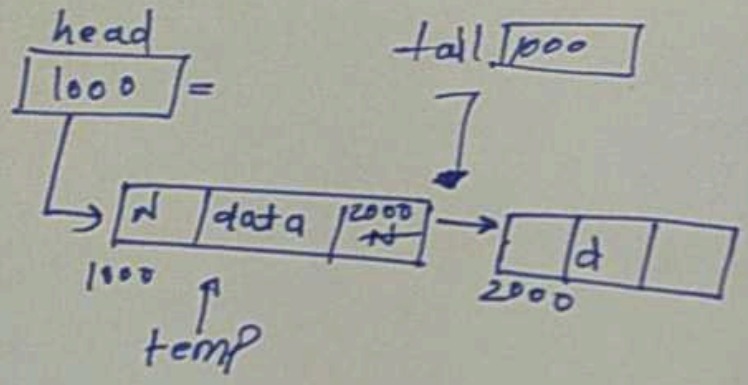
1] each process gets "dedicated address space".

- starting ~~VA~~ Virtual Address 0.
- usually impliment in paging.

Impoetants:-

STS (Shoet teem Schedulee).

when a peocess running on a CPU and suddenly a highee peioeity tayk comes then shoet teem schedullee peempte the running peocess take that into. Ready queue.



* ~~head → next = new node.~~
* ~~new node → pre =~~

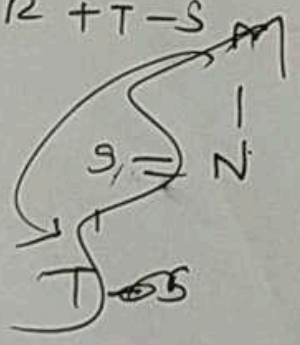
* temp → next = new_emp;
  new_emp → pre = temp;
* tail = new-emp

A + B A is mother of B
A − B A is brother B
A ÷ B A is father B
A * B A sister B

Q − N + M * P
~~P − S * N − Q~~
P + S * N − Q
✓ P − M + N * Q
Q − S ÷ P

A + B A is sister B
A * B A is wife B
A ÷ B A Father B
A − B A Bro B

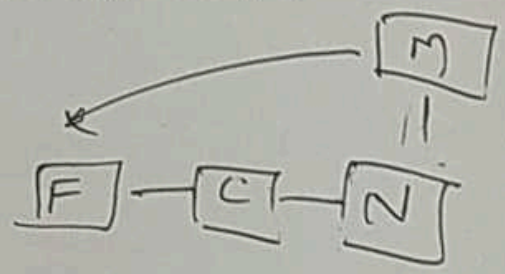
T daughter of ~~parent~~.
W
P * Q % R + S − T
W F S B
P * Q % R − T + S
W F B S
P * Q % R + T − S



A + B A is brother of B
A − B A is sister of B
A * B A is Father of B

C is son of M

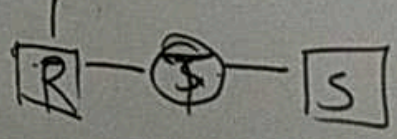
M − N * C + F
F − C + N * M
N + M − F * C
M * N − C + F



A + B A is Father B
A − B A is Bro B
A % B A is wif B
A * B A is moth B

M is materno germano mate T

M * N % S + T
M * N − S ÷ T
M * S − N + T
M * N * S % T

P * Q % R + S + T
M * N
S

Q = P