

Course Name : eDAC MAY-2021 (Pune & Karad)
Module Name : Operating System Concepts with Linux

16 Class Room Hrs + 8 Lab Hrs = 24 Hrs.

OS DAY-01:

- there are four steps to learn an OS subject:

1. **step1: "end user" -- linux commands** - user commands

2. **step2: "admin user" -- admin commands, shell script programming & installations.**

+ Responsibilities of System Administrator:

- installing and configuring softwares, hardware and networks.
- monitoring system performance and troubleshooting issues.
- ensuring security and efficiency of IT infrastructure.

3. **step3: "programmer user" -- system call programming**

4. **step4: "design/internals" -- to learn os internals/architecture**

+ Introduction to an OS:

Q. Why there is a need of an OS?

Q. What is Computer?

- Computer is a **machine/hardware/digital device** mainly contains: **Processor/CPU, Memory Devices, I/O Devices** etc....

- Basic Functions of Computer:

1. data storage
2. data processing
3. data movement
4. control

- by using computer machine various/different tasks can be performed efficiently and accurately.

- As any user cannot directly interacts with computer hardware, so there is a need of some interface between user & hardware, and to provide this interface is the job of an OS.

Q. What is a software?

- software is a collection of programs.

Q. What is a Program?

Program is a finite set of instructions written in any programming language (low level / high level), given to the machine to do specific task.

- There are 3 types of programs:

1. System Programs: programs which are built into an OS OR part of an OS.
e.g. Kernel, device driver, memory manager, loader, dispatcher etc...

2. Application Programs: programs which are used for specific purpose and which are not a part of an OS

e.g. MS Office, zoom, notepad, google chrome, calculator, games, task manager etc....

3. User Programs: programs defined by programmers

C, Java, C++ etc...

- It is not possible for any user to directly interact with an OS, and hence an OS provides 2 kinds of interfaces for the user in the form of program only. This program is referred to as a **shell**.

Shell - it is an application program through which a user can interact with an OS/Kernel.

- There are 2 types of Shell:

1. CUI (Command User Interface) Shell/CLI (Command Line Interface):

- in this type of interface a user can interact with an OS by means of entering commands in a text format through shell.

CUI shell accepts a command from the user in a text format and executes it from the kernel or gets a task done from the kernel/OS.

Linux: terminal - bsh, bash, csh, ksh etc...

Windows: command prompt - cmd.exe, powershell

MS DOS: command .com

2. GUI (Graphical User Interface) Shell

- in this type of interface a user can interact with an OS by means of making an event like left click, right click, double click, menu bar, click on buttons, etc....

Windows: explorer.exe (goto task manager -> search for process named as explorer.exe -> right click on it say end task => GUI will be disappeared).

run task : explorer.exe OR restart

Linux: GUI Shell Programs

GNOME: GNU Network Object Model Environment

KDE: K Desktop Environment

- We can install multiple OS's on single machine/HDD, but at a time only one OS can be active.
- VM Ware => Host OS & Guest OS
- bash, cmd.exe -> these application programs which either comes with an OS or can be installed later => **utility programs**

vs code editor - source code editor => editor specially used for programming

- **Eclipse is an IDE: Integrated Development Environment**, written mostly in Java and its primary use is for developing java applications, but it may be used to develop applications in other programming languages via plug-ins, including Ada, ABAP, C, C++, C#, COBOL, D, Fortran, JavaScript, Perl, PHP, Python, R, Ruby etc...

- An IDE is "**application software**" which is a collection of tools/programs like an **editor(source code editor)**, **compiler**, **linker**, **assembler**, **debugger** etc... required for faster software development.
- Any IDE normally consists of a **source code editor**, **build automation tools**, and a **debugger**.
- Most of the IDEs have intelligent code completion.
- Some IDEs such as **netbeans** and **eclipse**, contains a compiler, interpreter, or both.
- IDE is in contrast with vi editor, gcc, ld etc....

e.g. eclipse, netbeans, code blocks, visual studio, borland turbo c, turbo c++, android studio, anjuta, Dev C++, CodeLite, QT Creator etc...

Compilation Flow:

+ editor : it is an application program used for text editing also can be used to write a source code.

e.g. notepad, gedit, vi editor, eclipse source code editor program, vs code editor etc...

+ "preprocessor": it is an application program gets executed before compilation and performs two tasks:

1. removes all comments from the source code, and
2. executes all pre-processor directives like **#include**, **#define**, **#ifndef**, **#ifdef**, **#elif**, **#endif**, **#pragma** etc... conditional compilation preprocessor directive i.e. header guards.

e.g. "**cpp**" - c preprocessor, **M4 macro processor** in a UNIX like systems.

- the output of preprocessor is an **intermediate code**, as this file gets created with the combination of header file and source file, size of this file gets increases and hence it is also called as an **expanded source code**.

- command to create an intermediate code/file from source code/file:

`$gcc -E -o program.i program.c ==> program.i`

+ "**compiler**": compiler is an application program which converts high level programming language code (i.e. human understandable language code) into the low level programming language code (i.e. machine understandable language code), in our example from C programming language into an assembly language.

- output of a compiler is an "**assembly language code**".

- e.g. **GCC: GNU Compiler's Collection**, originally named as **GNU C Compiler**.

- **GNU: GNU's Not UNIX/GNU is Not UNIX** which is a recursive acronym.

- **GNU**: it is an Open Source Project by OSF (Open Source Foundation).

- Linux is also a GNU Project.

- Now a days GCC can be used for compilation of C++, FORTRAN, Objective C, Objective C++, Ada etc... programming languages.

- we need to use front ends of "gcc".

- Example: Borland's Turbo C, Turbo C++, Microsoft Visual C, etc...

- Compiler does **tokenization, syntax checking, code analysis, code optimization, parsing** etc...

"**Compiler's Construction**" -- By Aho, Ullman -- from AT&T Bells Labs.

- command to create an assembly language code from the source code:

`$gcc -S program.c ==> program.s`

+ **assembler**: it is an application program which converts assembly language code into the machine language code i.e. **object code**.

e.g. Linux: "**asm**"

Windows : Microsoft Assembler : "**masm**"

Turbo Assembler : "**tasm**"

etc...

- declarations of library functions exists in a **system header files** e.g. `stdio.h`, `string.h`, `stdlib.h` etc... , whereas definitions of all library functions are exists in a "lib" folder in a "**precompiled object module**" format.

+ **linker**: it is an application program which links object file(s) in a project with precompiled object modules of library functions and creates a "single" executable file.

e.g. "**ld**" -- link editor in Linux.

"**build = compilation + linking**"

compilation --> to create object code from source code.

linking --> linking of all object files with precompiled object modules by the linker and creates single executable file.

- command to create an executable file from object file:

`$gcc -o program.out program.o ==> program.out`

- command to execute a program:
\$/program.out

program.c ==> [preprocessor] ==> program.i ==> [compiler] ==> program.s/.asm ==> [assembler] ==> program.o/.obj ==> [linker] ==> program.out/.exe

What is a Process?

- process is program in execution
- running program is called as a process
- when a program gets loaded into the main memory
- program is a passive entity, whereas a process is an active entity.

High level: instructions written in this programming languages can be understandable directly by the programmer user.
C, C++, Java, Python etc....

Low level : instructions written in this programming languages can be either understandable directly by the machine or close to the machine.
e.g. assembly language

- In C programming language there are some features like pointers, register storage class etc... by using which we can go close to the machine and hence C programming language is also called as **middle level**.

Scenario-1:

Machine -1: Linux : program.c

Machine-2 : Windows : program.c ==> build (compile + link) & execute ==> YES

Portability: program written in C on one machine/platform can be compile and execute on any other machine/platform.

Scenario-2:

Machine -1: Linux : program.c ==> build (compile + link) ==> program.out (executable code)

Machine-2 : Windows : program.out (executable code) ==> execute ==> NO

function or method or routine or procedure or algorithms or events/operations

Q. Why an executable file/code created on one machine/platform cannot be execute on any other machine platform?

- In Linux, file format of an executable file is **"ELF": Executable & Linkable Format (Formerly named as Extensible Linking Format)**, whereas in Windows, file format of an executable file is **"PE": Portable Executable**.
- An ELF is a common standard file format for **an executable files, object codes, shared libraries, and core dumps**.
- **core dump** is also called as **crash dump, memory dump, or system dump** consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.
- An executable file is a program only, that contains set of instructions with extra information.
- In Linux, ELF is a **"sectioned binary"**, i.e. executable file in linux is divided logically into different sections, and each section contains specific data.
- An elf file mainly it contains: **exe header/primary header/elf header, data section, bss section i.e. block started by symbol section, rodata i.e. read only data section, code/text section, symbol table** etc....

1. **"exe header/primary header"**: it contains info to starts an execution of a program (executable file), mainly it contains:

i. **Magic Number**: it is a constant number generated by the compiler which is file format specific.

- In ELF magic number starts with **"7f E L F"** -- ASCII values of letters E, L & F in its equivalent hexadecimal format.

- In PE -- magic number starts with ASCII values of letters M Z in its equivalent hexadecimal format, as **"Mark Zbikowski"** architect of Windows operating system at Microsoft.

ii. **addr of an entry point function**: it contains an addr of **_start() routine** in which addr of function can be kept from which an execution to be started. By default it contains an addr of **main() function** and it can be changed as well.

iii. **info about remaining sections (metadata -- data about data)** etc....

2. **"data section"**: it contains initialized global & static variables.

3. **"bss section (block started by symbol)"**: it contains uninitialized global & static variables.

4. **"rodata section (read only data section)"**: it contains constants & string literals.

e.g.

constants => 1000, 100L, 0x42, 012, 'A', 4.5f, 10.20 etc...

string literals => char *str = "sunbeam infotech";

5. **"code/text section"**: it contains executable instructions

6. **"symbol table"**: it contains info about symbols i.e. functions and their variables in tabular format.

etc...

+ Magic Number:

- Magic number are the first few bytes of a file which are unique to a particular file type. These unique bytes are referred as **magic number**, also sometimes referred as **file signature**.
- These bits/bytes can be used by the system to **differentiate between and recognize different files without file extension** (specially in Linux/UNIX based OS).

```
sunbeam@sachin-lenovo:~/feb2021/eDAC/OS/project$ size program.out
text  data      bss  dec      hex      filename
1754      616          8  2378      94a      program.out
```

intialized global & static variables:

```
sunbeam@sachin-lenovo:~/feb2021/eDAC/OS/project$ size program.out
text  data      bss  dec      hex      filename
1754      624          8  2386      952      program.out
```

unintialized global & static variables:

```
sunbeam@sachin-lenovo:~/feb2021/eDAC/OS/project$ size program.out
text  data      bss  dec      hex      filename
1754      616         16  2386      952      program.out
```

- **loader**: it is a system program (i.e. inbuilt program of an OS), which loads an executable file from HDD into the main memory.

- When we execute a program, loader first verifies file format of an executable file, if file format matches then it checks magic number, and if file format as well magic number both matches then only it loads an executable file from hdd into the main memory => an execution of a program is started OR process has been submitted.

What is an OS?

- It is a **system software** (i.e. collection of system programs), which acts as an **interface between user and computer hardware** (i.e. processor , memory devices & io devices etc...).
- An OS also acts as an **interface between programs (user & application programs) and computer hardware**.
- As an OS allocates required resources like **CPU time, main memory, i/o devices access** etc... to running programs, it is also called as "**resource allocator**".
- As an OS manages limited available resources among all running programs, it is also called as a "**resource manager**".
- An OS provides environment/platform to all types of programs to complete their execution.
- An OS controls execution of all programs, as well as it controls all h/w devices connected to the computer system, so it is also called as a "**control program**".

- An OS is a **software** (i.e. collection of hundreds of system programs and application programs in a binary format) which comes in either CD/DVD/PD, mainly having three components:

OS = Kernel + Utility Programs + Application Programs.

1. Kernel: it is a core part/program of an OS that runs continuously into the main memory does basic minimal functionalities of it.

i.e. Kernel = OS - without kernel OS is nothing

- "Kernel is OS OR OS is Kernel".

2. Utility Programs/Softwares:

- e.g. disk manager, task manager, anti-virus software, explorer.exe, backup software etc...

3. Application Programs/Softwares:

- e.g. MS Office, vi editor, etc....

- Linux Kernel - its source code is freely available onto the internet => **open source OS, specially designed for R&D.**

- Windows Kernel source code is not freely available - commercial OS specially designed for business.

+ Total there are 8 functions of an OS:

kernel /compulsory / core functionalities/basic minimal functionalities:

1. Process Management
2. Memory Management
3. File & IO Management
4. Hardware Abstraction
5. CPU Scheduling

extra utility/optionl functionalities

6. User Interfacing
7. Networking
8. Protection & Security

Installation of an OS:

- To install an OS onto the computer machine, is nothing but to store OS software (i.e. collection of thousands of system programs & application programs which are in a binary format/object code format) onto the **HDD**.

Bootting:

- If any OS wants to becomes active, atleast its core program i.e. kernel must be loaded intially from HDD into the main memory, and process of loading of kernel of an OS from HDD into the main memory is called as a **bootting** and it is done by **bootstrap program**.

- bootstrap program is in HDD, which is invoked by **bootloader program**.

- **bootloader program resides into the HDD**

- While booting kernel of an OS gets loaded into the main memory and runs continuously into it does basic minimal functionalities, and kernel remains present inside the main memory till we do not shutdown the system.

core computer system: components which are mounted/attached onto the motherboard onto the slots is referred as core computer system.
e.g. CPU, Main Memory, Cache Memory, ROM etc...

peripheral devices/peripherals/external devices: devices which are connected to the motherboard externally through ports.
e.g. keyboard, monitor, mouse, hdd etc...

- **bootable device/partition:** if any **storage device/partition** contains one special program called as **bootstrap program** in its first sector i.e. in a **boot sector (first 512 bytes in a storage device/partition)**, then such a device/partition is referred as bootable device/partition.

- if boot sector of any storage device do not contains bootstrap program then it will be referred as non-bootable.

There are 2 stpes of booting:

1. machine (h/w) boot:

step-1: when we swicthed on power supply, current gets passed to the **motherboard** on which **ROM Memory** is there, which contains one micro-program called as **BIOS (Basic Input Output System)** gets executes first.

step-2: first step of BIOS is **POST (Power On Self Test)**, under POST it checks wheather all peripheral devices are connected properly or not and their working status.

step-3: after POST, BIOS program invokes **bootstrap loader program**, which searches for available bootable devices in a computer system, and it selects only one bootable device at a time as per the priority decided in a BIOS settings. (bydefault it selects HDD as a bootable device).

2. system boot:

step-4: after selection of HDD as a bootable device, **bootloader program** which is present inside it gets invokes, and it displays list of names operating systems installed onto the HDD, from which user need to select any one OS at a time.

step-5: upon selection of any OS, **bootstrap program** of that OS gets invokes and it locates kernel into the HDD and load it into the main memory.

- We can install multiple OS's on single HDD, but to do this we need to divide HDD logically into the partitions and on each partition we can install one OS.

OS DAY-02:

- UNIX
 - Windows
 - Linux
 - MAC OS X
 - Android
 - Solaris
 - Symbian
 - Palm OS
 - Google OS
 - Kali Linux
- etc....

UNIX (UNICS): Uniplexed Information & Computing Services/System

Q. Why UNIX?

- OS Concepts with Linux
 - Linux is UNIX like OS i.e. Linux OS was developed by getting inspired from UNIX OS.
 - Whatever is there in UNIX, everything is there into the Linux.
 - **UNIX OS is treated as mother of all modern OS's** like : Linux, Windows, MAC OS X, iOS etc..., as all these OS's follows system arch design of UNIX.
 - UNIX was specially designed for developers by the developers.
 - UNIX was developed @AT&T Bell Labs in US in the decade of 1970's by Ken Thompson, Denies Ritchie & team.
 - In UNIX first time, **multi-tasking, multi-threading & multi-user** feateures has been introduced.
 - UNIX is the most secured OS and hence Linux
 - Filesystem of UNIX is most secured, files of one user can be protected from another users by giving perms.
 - UNIX & Linux mostly used for Server machines.
-
- Linux OS was developed by **Linus Torvalds in University Helsinki, in the decade of 1990's, and first linux kernel (vimlinuz) version 0.01 was released in the year 1991.**
 - Linux is an open source OS: source code of linux kernel is freely available onto the internet, so anyone can download it and can add its own utilities and come up with their of distro of Linux
 - Ubuntu Linux
 - Fedora Linux
 - Centos Linux – Embeded Developement
 - RedHat Linux
- etc...
- Android (Java) is Linux based OS used in mobile smart phones.

+ System architecture of UNIX

- Kernel acts as an interface between user programs & computer hardware.

- Human Body System:

- respiratory system
- digestive system
- nervous system
- reproduction system
- etc...

- for functioning of whole human body system all subsystems work together, similarly, in any OS, there are diff subsystems which work together for functioning whole OS.

- An OS contains following subsystems:

- file subsystem
- process control subsystem: ipc, memory management & scheduling
- system call interface block
- buffer cache
- character devices & block devices block
- device drivers
- hardware abstraction block
- etc...

- there are 2 major subsystems of any OS:

1. process control subsystem
2. file subsystem.

- file & process are two very important concepts of any OS.

- In UNIX, "file has space and process has life"

- In UNIX, whatever that can be stored is considered as a file, whereas whatever is active is considered as a process.

- KBD => hardware / input device => file

- Monitor => hardware / output device => file

- HDD => hardware / memory device => file

UNIX categorised devices into two categories:

1. **character devices:** devices from which data gets transferred char by char i.e. byte by byte

UNIX treats character devices as **character special device files (c)**.
e.g. keyboard, monitor, printer, mouse etc....

2. **block devices:** devices from which data gets transferred **block by block** i.e. sector by sector (usually size of 1 sector = 512 bytes).

UNIX treats block devices as **block special device files (b)**

e.g. all storage devices like HDD, PD, CD, DVD etc...

When we copy data from HDD to PD

- UNIX treats this as a transfer of data from one file to another file.

- In UNIX / Linux there are 7 types of file:

1. **regular file (-)** : e.g. text files, image files, audio files, video files, source files etc...

2. **directory file (d)** : UNIX treats all directories as well dir file whose contents are names of files & sub dir's in it.

3. **character special device file (c)**: all character devices

4. **block special device file (b)**: all storage devices

5. **linkable file (l)** : it is used to store link of existing file its a symbolic link (like in windows – shortcut of any file which is onto the desktop).

6. **pipe file (p)**: this type of file is used in inter process communication in pipe command & pipe() system call.

7. **socket file (s)**: this type of file is used in inter process communication across the systems.

Human Body + Soul => Active => Process
Human Body - Soul => Passive => Dead Body => File

- file has space & process has life

- **device driver**: it is system program/s which enables the kernel/system to communicates with that particular hardware/device.

- **buffer cache** – it is a purely software technique, in which kernel used **portion of the main memory** to keep/store most recently accessed **disk** contents to achieve max throughput in min hardware movement.

- hardware control block is used by the kernel for hardware abstraction.

- L1 cache & L2 cache (level -1 & level-2 cache) its hardware, cache memory used to reduce speed mismatch between the CPU and cache memory.

- L1 cache => data cache

- L2 cache => instruction cache

- In modern CPU's cache memory is inbuilt into it.

- file & io management, process management, ipc, scheduling, memory management, hardware abstraction are services/functionalities of the kernel.

Kernel = Program

functions: system calls:

fork()
open()
read()
write()
.
.
.

Linked List: slll.c

main(); => client

services:

create_node();
add_last();
add_first();
add_at_pos()
delete_last();
delete_first();

- **system calls**: are the functions defined in C, C++ & assembly language which provides an interface to the services made available by the kernel for user (programmer user).

System call interface block => system calls API's : Application Programming Interface

- If any user (programmer user) wants to use services made available by the kernel in his/her program, then either it can be used directly by means of giving call to **system call API's** or can be used indirectly by means of giving call to library functions from inside which there is call given to system call API's only.

Wrapper functions => are used just for calling original functions.

file handling:

C Programming:

fopen() lib function internally makes call to **open()** sys call api, which internally makes call to **_open()** sys call.

fopen() => **open()** sys call api : to open a file / to create a new file

`fwrite()/fprintf()/printf()/fputs()/fputc() => write() sys call api => _write() system call.`

`fread()/fscanf()/scanf()/fgets()/fgetc() => read() sys call api => _read() sys call.`

- In UNIX there are 64 system calls
- In Linux there are 300 system calls
- In Windows there are more than 3000 system calls

In UNIX => `fork()` : to create a new process / child process

In Linux => `fork()` / `clone()` / `vfork()` : to create a new process / child process

In Windows => `CreateProcess()` : to create a new process / child process

- Irrespective of an OS there are 6 categories of system calls:

1. **file manipulation/operations system calls:** e.g. `open()`, `write()`, `read()`, `close()`, `lseek()` etc...

2. **device control system calls:** e.g. `open()`, `write()`, `read()`, `close()`, `ioctl()` etc...

3. **process control system calls:** e.g. `fork()`, `_exit()`, `wait()` etc...

4. **accounting information system calls:** e.g. `getpid()`, `getppid()`, `stat()` etc...

5. **inter process communication system calls:** e.g. `pipe()`, `signal()`, `kill()` etc...

6. **protection & security system calls (filesystem level) :** `chmod()`, `chown()` etc...

```
//user defined code/user program
#include<stdio.h>
int main(void)
{
    int n1, n2;
    int res;

    printf("enter n1 & n2: "); //write( ) sys call - system defined code
    scanf("%d %d", &n1, &n2); //read( ) sys call - system defined code

    res = n1 + n2; //arithmetic expression
    printf("res = %d\n", res); //write( ) sys call - system defined code

    return 0; //successful termination
}
```

sum.c => build => sum.out => execute

- Whenever system call gets called the CPU switches from user defined code to the system defined code, and hence system calls are also called as **software interrupts/trap**.

What is an interrupt?

- an interrupt is a signal which received by the CPU from any io device due to which it stops an execution of one job/process and starts executing another job/process => **hardware interrupt**.

- throughout an execution of any program, the CPU switches between user defined code and system defined code, and hence we can say system runs in a two modes: kernel mode & user mode, and this mode of operation of the system is called as **dual mode operation/dual mode protection**.

1. kernel mode: when the CPU executes system defined code instructions, we can say system runs in a kernel mode/system mode.

- kernel mode is also called as **system mode/protected mode/privileged mode/monitor mode**.

2. user mode: when the CPU executes user defined code instructions, we can say system runs in a user mode.

- user mode is also called as **non-privileged mode**

- The CPU can differentiate between user defined code instructions and system defined code instructions by using one bit which is onto the CPU called **mode bit** maintained by an OS.

mode bit = 0 => kernel mode

mode bit = 1 => user mode

- Whenever system call occurs, an OS clears the mode bit (to make the value of bit as 0), after completion of system call value of mode bit will be set (to make the value of mode bit as 1).
- The CPU can have access of hardware devices only in a kernel mode, access h/w devices is restricted for the CPU in a user mode, protection of h/w devices can be achieved.

user written one program => there is logic to crash hdd

the CPU is currently executing user defined code => mode bit = 1 => user mode.

=> There are major subsystems of any OS:

1. Process Control Subsystem
2. File Subsystem

OS DAY-03:

System Arch Design of UNIX:

- file subsystem
- process control subsystem

1. Process Control Subsystem:

Q. What is a Program?

User Point of view:

A Program is a finite set of instructions written in any programming language given to the machine to do specific task.

System Point View:

Program is nothing but an executable file onto the HDD, which has got elf header, bss section, data section, rodata section, code section, symbol table etc...

- Program is a passive entity, whereas process is an active entity

Q. What is a Process?

User Point View:

- Program in execution
- Running instance of a program is called as a process
- When a program gets loaded into the RAM/Main Memory it becomes a process.

System Point of View:

- Process is nothing but program (an executable file) which is into the main memory has got one structure i.e. PCB for it into the main memory inside kernel space and program is there inside user space has got:

bss section, rodata section, code section and 2 new sections got added for a process by the kernel/OS:

stack section: it contains FAR's of called functions

heap section: dynamically allocated memory

Q. Why RAM is called as Main Memory?

Ans: For an execution of any program RAM memory is must, and hence it is also called as Main Memory.

Kernel: it is a core program/part of an OS which runs continuously into the main memory and does basic minimal functionalities it.

- kernel gets loaded into the main memory while booting, and it resides always into the main memory till we do not shutdown the system, and hence few part/portion of the main memory will be always occupied by the kernel.

- main memory is divided logically into two parts:

1. **kernel space:** portion of the main memory which is occupied by the kernel

2. **user space:** portion of the main memory other than kernel space

- When we execute a program (either by double clicking on icon of an executable file OR ./program.out + press enter):

- loader very first verifies file format, if file format matches then it checks magic number, if file format as well as magic number both matches then only it starts an execution of that program/process gets submitted.

- When we say process gets submitted/an execution of a program is started => practically speaking : PCB gets created for that program/process:

upon process submission:

step-1: one structure gets created by the kernel for that process into main memory inside kernel space referred as **PCB (Process Control Block)**, PCB contains all the information which is required to complete an execution of that program.

step-2: loader removes elf header & symbol table from program, and it copies bss section, data section, rodata section & code section as it is into the main memory inside user space and 2 new sections got added for the process:

1. stack section

2. heap section

- Upon process submission, PCB for that process gets created into the main memory inside kernel space and it remains presents there till process do not gets terminated, i.e. when an execution of program is completed PCB of that process gets destroyed/removed from the main memory.

no. of PCB's into the main memory = no. of processes running in the system

- Per process an OS creates one structure referred as PCB. PCB is also called as PD(Process Descriptor), In Linux PCB is referred as TCB(Task Control Block).

- PCB/TCB/PD, **mainly it contains:**

1. **pid** – process id – unique identifier of a process for OS
2. **ppid** – parent's process id
3. **PC: Program Counter** - it contains an addr of next instruction to be executed
4. **Memory Management information**
5. **CPU Scheduling Information** like priority of a process, cpu sched algorithm etc...
6. Information about resources allocated for that process
7. Information about IO devices allocated for that process
8. **Execution Context:**

When the CPU is currently executing any program, information about data & instructions of that program can be kept/store temporarily into the CPU registers, collectively this information is referred as an execution context. Copy of an execution context also can be kept into the PCB of a process.

9. **current state of the process:** new state / ready state / running state / waiting state / terminated state.

etc.....

- In Linux size of PCB is around more than 1 KB

shell => systemd – system daemon process

systemd => init process

init process – self generated process

upon process submission => PCB is there for a program into the main memory inside kernel space => **running program**.

after process submission => it may be active or inactive

active running program => if PCB is there into the main memory inside kernel space and program is also there into the main memory inside user space.

inactive running program => if PCB is there into the main memory inside kernel space but program is not there into the main memory inside user space (it can be kept temporarily into swap area).

if PCB is not there into the main memory inside kernel space => program is not running i.e. an execution of a program is finished.

- at a time multiple programs can be active, but there is limit on no. of processes can be active at a time.

- Throughout an execution of any program it goes through different states and at a time it may exist only in one state, and current state of a process can be stored into its PCB.

- there are total **5 states of process**:

1. new state: upon process submission it is considered in a new state.

i.e. if PCB for any process gets created into the main memory inside kernel space state of that process is considered as a new state.

2. ready state: after process submission, if process is there into the main memory and waiting for the CPU time i.e. ready to run onto the CPU, state of that process is considered as ready state.

3. running state: if the CPU is currently executing any process, state of that process is considered as running state.

4. waiting state: if a process is requesting for any io device, then it goes into waiting state.

After io request completion process changes its state from waiting to ready

5. terminated state: after exit process goes into the terminated state

i.e. if PCB of a process is destroyed from the main memory it is considered in a terminated state.

dispatcher: it is a system program (i.e. inbuilt program of an OS) which loads program (i.e. data & instructions of a program) from the main memory onto the CPU registers.

eDAC + eKDAC = DAC

Batch => logical

Physical => each & every student

process - logical

thread - practical

- thread - smallest indivisible part of a process

- thread smallest execution unit of a process.

+ Features of an OS:

1. Multi-Programming: system in which multiple jobs/processes can be submitted at a time, i.e. at a time an execution of multiple i.e. more than one programs can be started.

- at a time multiple PCB's can be created into the system.

- **degree of multi-programming** - no. of processes that can be submitted into the system at a time.

2. Multi-Tasking: system in which the CPU can execute multiple processes concurrently / simultaneously (i.e. one after another).

i.e. The CPU can execute only one process/job at a time.

- the CPU executes multiple processes concurrently with such great speed, we feel/it seems that, CPU executes multiple processes at once, and hence this feature is referred as multi-tasking.

- **Multi-tasking** is also referred as **time-sharing** => system in which the CPU time gets shared among all running programs.

Eskon temple => arati

group of people for performing "arati"

panditji - 1 => process-1

panditji - 2 => process-2

panditji - 3 => process-3

panditji - 4 => process-4

- there are two types of multi-tasking:

1. process based multi-tasking => fork() system call

2. thread based multi-tasking => multi-threading programming with pthread library is used.

person=1 =>

3. Multi-Threading: system in which the CPU can execute multiple threads which are of either same process or are of different processes concurrently/simultaneously.

- the CPU executes multiple threads/processes concurrently with such great speed, we feel/it seems that, CPU executes multiple threads of either same process or are of diff processes at once, and hence this feature is referred as multi-threading.

- CPU time gets shared among multiple threads of all processes.

"The CPU can execute only one thread of any one process at a time".

4. Multi-Processor

5. Multi-User

- Drive Car:

DAY-01:

step-1: switch on

step-2: press clutch fully & break fully

step-3: we need to change gear from neutral to 1st

step-4: release clutch gradually till half

step-5: release break gradually

step-6: slowly increase accelerator as per requirement

.
. .
. .
. .

#DAY-20:

step-1: switch on

step-2: press clutch fully & break fully

step-3: we need to change gear from neutral to 1st

step-4: release clutch gradually till half

step-5: release break gradually

step-6: slowly increase accelerator as per requirement

.
. .
. .
. .

- we feel that we performed all steps at once => ??

responsiveness to stimuli => reaction time given by the brain to all actions is so quick, we cannot differentiate all our actions.

- **google chrome** => process => thread based multi-tasking

new tab -1 => new thread gets created

new tab -2 => new thread gets created

- **mozilla firefox** => process => process based multi-tasking

new tab -1 => new diff process gets created

new tab -2 => new diff process gets created

google chrome => youtube

downloading => thread1

video streaming => thread2

audio streaming => thread3

- you dont have to programming by using vi editor
vs code editor

pid_t fork(void);

fork() sys call createa new process/child process by duplicating calling process, which do not accepts anything and it returns pid of child process to its parent and 0 into the child process.

pid is of type of **unsigned int**

typedef unsigned int pid_t;

ret = fork();//function call

ret = return value of fork() is getting assigned to ret
after fork() system call execution statement, assigning return value of fork() to ret var

OS DAY-04:

`exit()` C lib function internally makes call to `_exit()` system call, due to which calling process gets terminated.

```
void _exit(int status);
```

`_exit()` system call terminates calling process, while calling this `_exit()` we need to pass exit status of type `int`.

exit status with exit value 0 => successfull termination

```
_exit( 0 );//successfull termination
```

exit status with exit value <0 => abnormal termination

```
_exit( -1 );//abnormal termination
```

e.g. divide-by-zero error - process gets aborted

exit status with exit value >0 => erroneuos termination

```
_exit( 1 );//errorneous termination
```

e.g. if we are trying to open a file in readonly purpose and file is not exists
`malloc()` function failed due to insuff heap memory

- In a C program, can we execute statement/s inside if block as well as else block.

=> `fork()`; //its special function which returns more than one values

```
int i ;
```

```
fork( ) => process based multi-tasking
```

```
if( num == 0 )
```

```
{
```

```
    //statement/s
```

```
}
```

```
else
```

```
{
```

```
    //statement/s
```

```
}
```

tty: tele type terminal:

terminal => console input and console output

CUI terminal => core terminal => tty

GUI terminal => terminal which is running on GUI shell is not a core terminal
it is referred as pseudo terminal (pts)

- how many terminals

- there are 2 types of shell programs:

CUI shell

GUI shell =>

- if we want to goto core shell/core CUI shell tty => cntrl+alt+F2(Function Key) if required)/cntrl+alt+F3/

+ Comamnd Name : logout

user can logout from his/her account

Q. What is difference between ps & top?

top as well ps are to display information about running programs:

top:

ps

- to achieve process based multi-tasking by using fork() system call:

- orphan process

- zombie process

thread based multi-tasking:

decrement the value of counter => child process
increment the value of counter => process process
- parent & child are running concurrently

decrement the value of counter => within the same process -> thread
procedure : dt - there is need to attach dt with thread procedure implemented
for decrement counter

increment the value of counter => within the same process -> thread
procedure : it - there is need to attach it with thread procedure implemented
for increment counter.

dt & it => threads executed concurrently within same process

`void *malloc(size_t size);`

`malloc()` is used to allocate memory dyanmically to store any type of elements
its resposibility of the programmer to typecast return value/addr of `malloc()`
function into the required type of pointer.

`void` => nothing

`void *` => generic pointer - anything

//we have define thread procedure by following below signature:

`void *funtion_name(void *);`

this function : returns any type of value & any no. of values, whereas it accepts
any type of arguments and any no. of arguments.

Multi-Programming

Multi-Tasking: process based & thread based (Multi-Threading)

Multi-User

Uni-Processor: System can run on such a machine in which only one CPU is there.

e.g. MS DOS, Windows Version before **Windows Vista**

Multi-Processor: System can run on such a machine in which more than one CPU's/processors are connected in a closed circuit.

e.g. **Windows Vista & onwards**

Linux 2.5+ versions

_fork() => core system call

fork() => system call api provided by the kernel for application programming

fork() => programmer user fork() system call