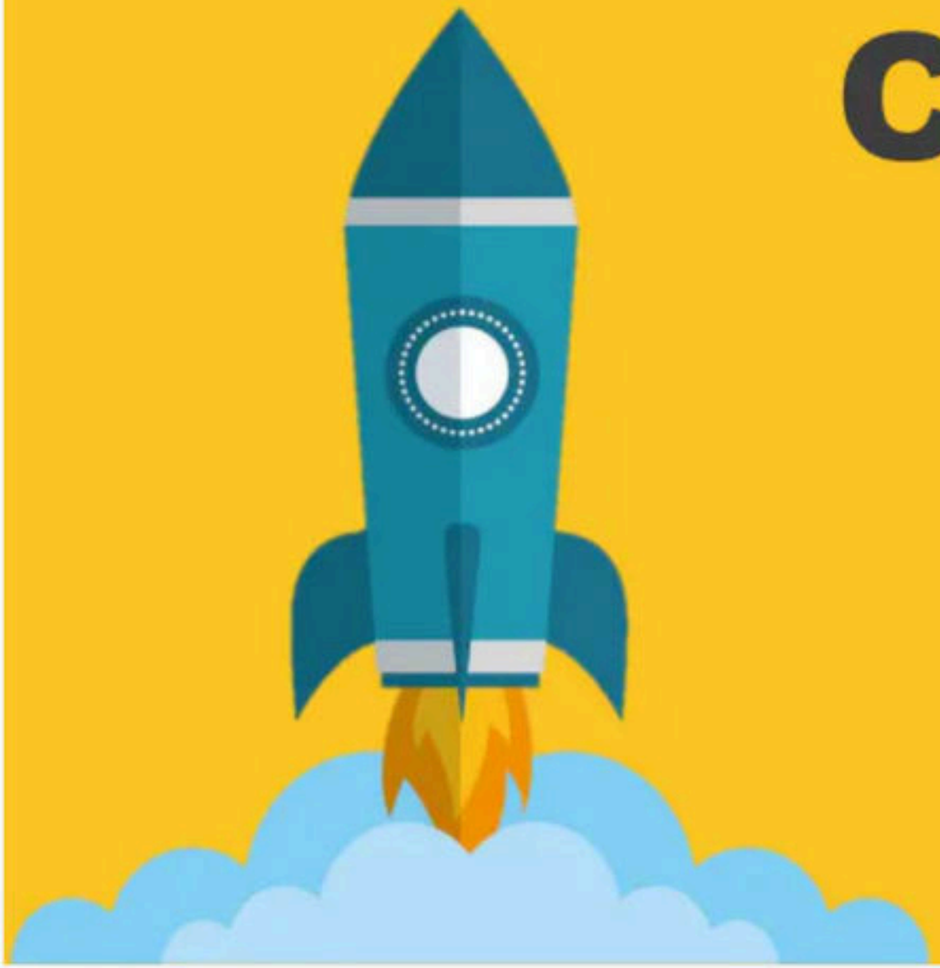
LAUNCHING LINUX COURSE



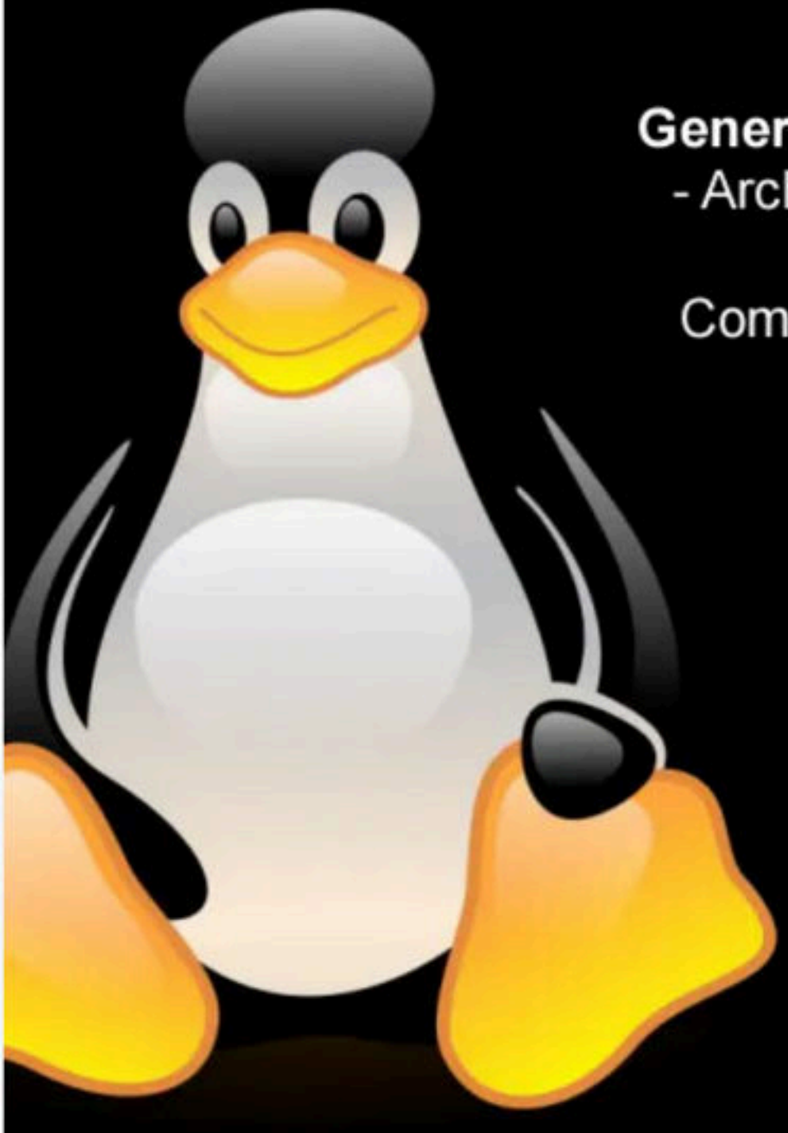
HELPFUL FOR

- All Engineering Students who are Pursuing their B.E/B.Tech which will help in their Semester Exam Preparation.
- Students who are Preparing for UGC NET
- Working Professionals

WE WILL BE LAUNCHING MANY MORE SUBJECTS SOON

All these Subjects which I am going to Launch will be taught in the following manner

- Concepts Covering Fundamentals to Deep Areas.
- Practice & Interview Questions
- Practical Execution of the Programmes.



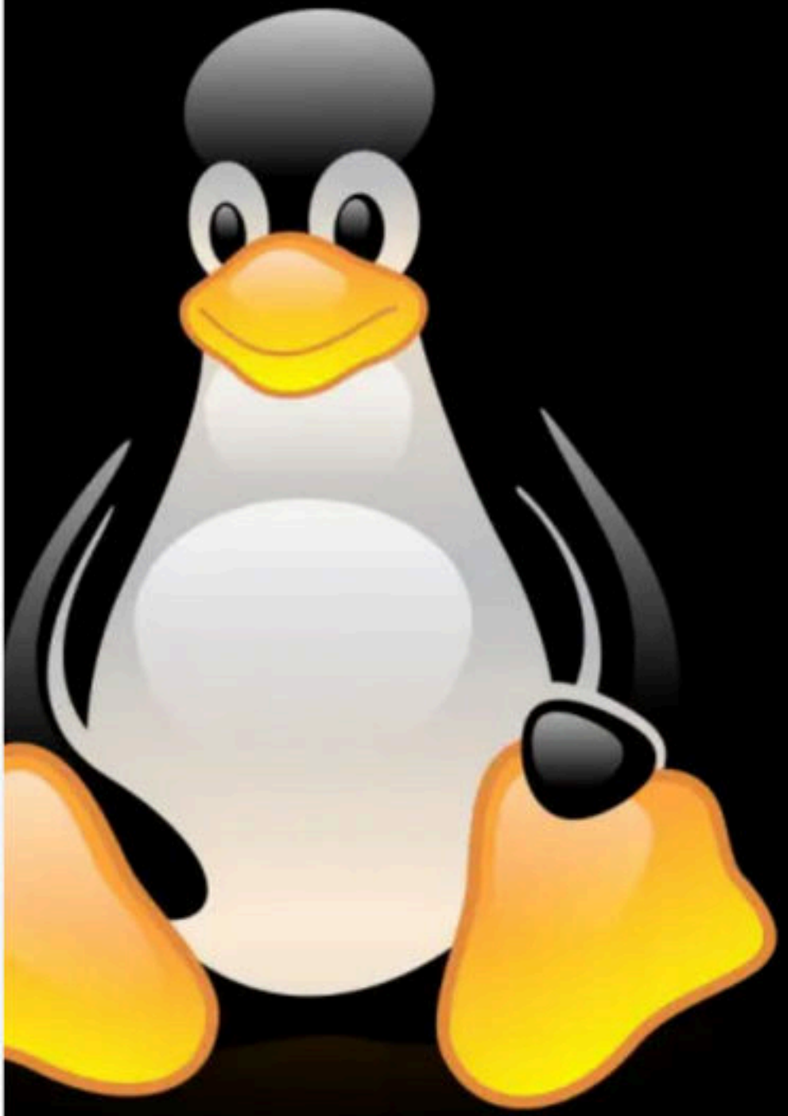
Topic 1: Introduction

General Overview: History of Linux and comparison with windows - Architecture of Linux operating system - Introduction to system concepts - General features of Unix commands – Command arguments and options - Exploring basic commands

Topic 2: LINUX Files

Internal Representation of Files: Inodes – Structure of a Regular File – Directories – SuperBlock – Inode Assignment to a New File – Allocation of Disk Blocks

File manipulation : Open, Read, Write to a file - Changing file/directory attributes **-** Mounting And Unmounting File Systems - Usage of commands related to files/directories - system calls performing operations on files.



Topic 3: Shell programming

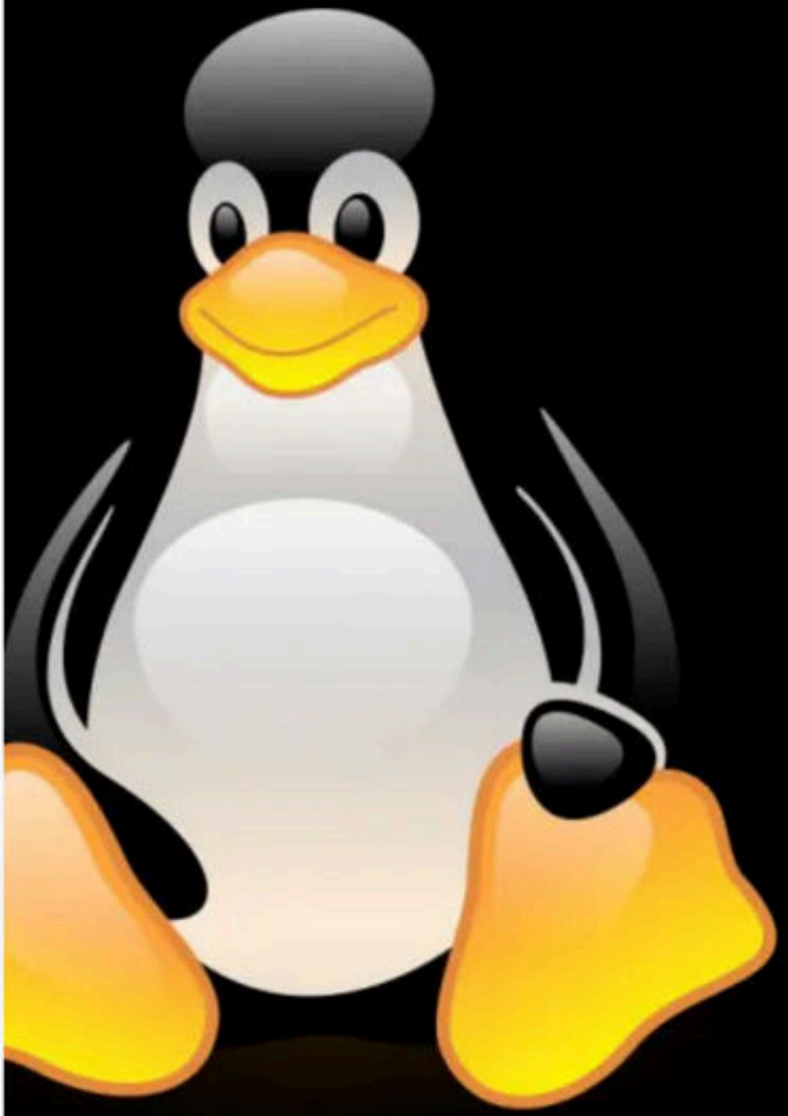
Ordinary variables and environment variables - Command interpreters and the .profile file - Command line arguments - Logical operators for conditional execution - The if, while, for and case control statements - Example shell programs

Topic 4: LINUX processes

Process States and Transitions – The Context of a Process - Memory layout of a program - Shared Libraries - Memory Allocation - Environment Variables.

Process control : Process Creation – Awaiting Process termination – Invoking other Programs – Changing the size of a Process - Process Scheduling - Race Conditions

Inter process communication: Messages - Shared memory - Semaphores



Topic 5: Daemon Processes

Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model.

Topic 6: Buffer cache

Buffer headers – Structure of the Buffer Pool – Scenarios for Retrieval of a Buffer– Reading and Writing Disk Blocks – Advantages and Disadvantages of the Buffer Cache.

System info related commands:-

	Command	Meaning of the command
1	date	Displays the current date and time
2	cal	Shows the current month's calendar
3	uptime	Displays the current uptime of the system
4	whoami	Displays the current logged in username
5	uname -a	Shows the kernel information
6	cat /proc/cpuinfo	Displays CPU info
7	cat /proc/meminfo	Displays memory info
8	df -h	Shows the disk usage of the system in human readable form
9	du -h	Shows the current directory space usage in human readable form
10	whereis cmd	Shows the possible locations of the command cmd

File management commands:-

	Command	Meaning of the command
1	pwd	Show the present working directory
2	Ls	List all the files in the current directory
3	ls –a	List the files in the current directory, even the hidden files
4	ls –t	List the files sorted by time of modification
5	cd DIR	Change directory to folder named DIR
6	Cd	Change to home directory
7	mkdir NEW_FOLDER	Create a directory named NEW_FOLDER
8	rmdir NEW_FOLDER	Deletes the directory named NEW_FOLDER
9	more FILE1	Prints the contents of FILE1
10	head FILE2	Prints the first 10 lines of FILE2

	Command	Meaning of the command
11	tail FILE3	Prints the last 10 lines of FILE3
12	touch FILE4	Creates an empty document with name FILE4
13	rm FILE5	Deletes the document named FILE5
14	rm -r DIR	Deletes the directory DIR
15	cp FILE6 FILE7	Copy the contents of FILE6 to FILE7
16	mv FILE8 FILE9	Rename or move FILE8 to FILE9
17	find FILE10	Search for the FILE10 in the current directory
18	wc FILE11	Prints the number of lines, words and byte counts from the FILE11
19	chown user FILE12	Change the owner of FILE12 to ‘user’
20	chgrp grp FILE13	Change the group ownership of FILE13 to ‘grp’

Process management commands:

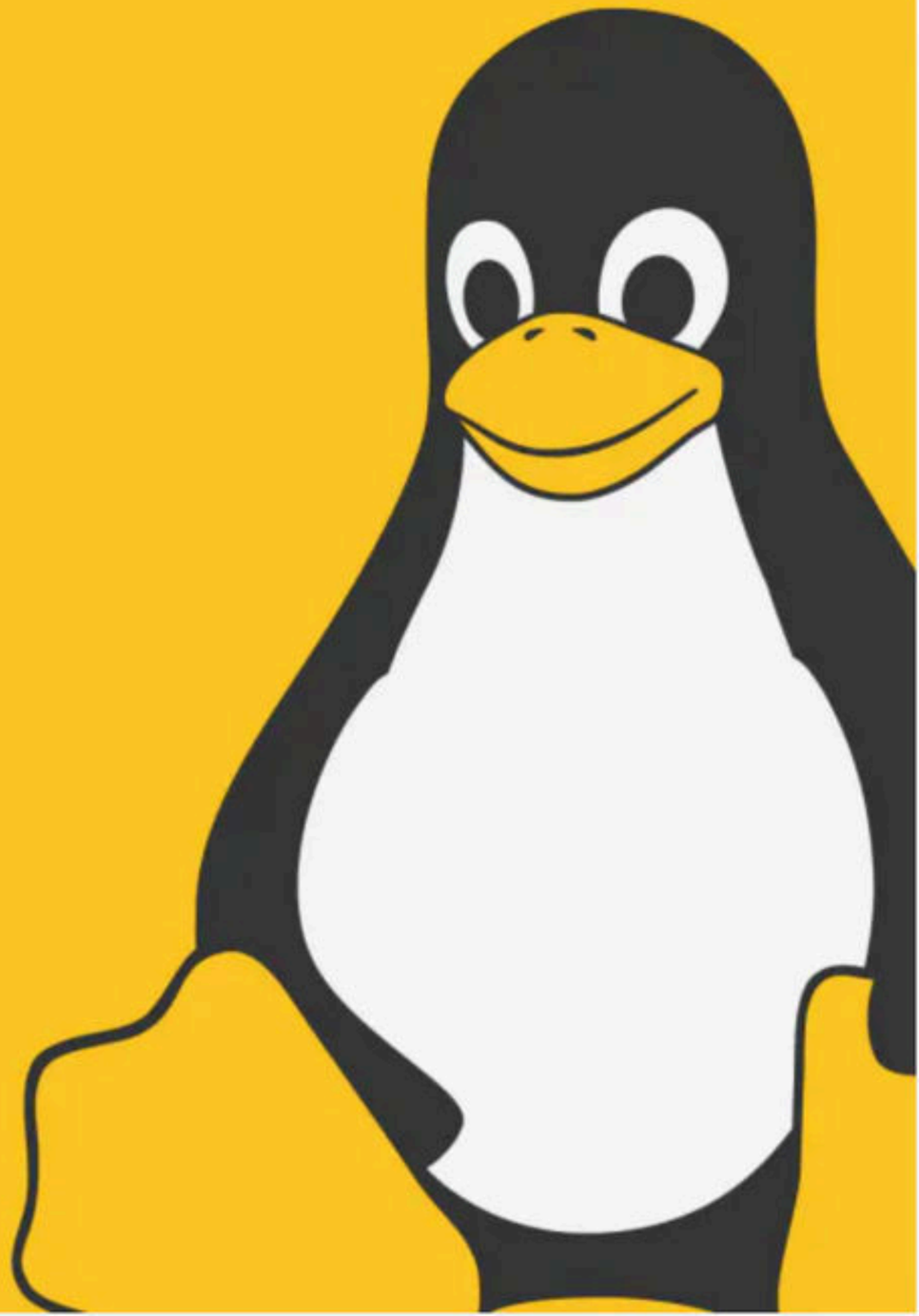
	Command	Meaning of the command
1	ps	Displays the currently working processes
2	top	Displays the dynamic real-time view of the running system and also prints the list of currently running processes
3	kill PID	Kills the process with the given process id
4	killall PROC	Kills all the processes having the name PROC
5	pkill PATTERN	Kills all the processes matching the PATTERN
6	jobs	Prints the list of jobs that the current shell is managing.
6	bg	Lists stopped or background jobs, resumes a stopped job in the background
7	fg	Brings the most recent job to the foreground
8	fg n	Brings job with the number n to the foreground

Network commands:

	Command	Meaning of the command
1.	ping host	Ping host and output results
2.	whois domain	Get whois information for domains
3.	dig domain	Get DNS information for domain
4.	dig -x host	Reverse lookup host
5.	wget file	Download file
6.	wget -c file	Continue a stopped download

LINUX
LECTURE 1

Linux History
Linux vs Windows
Linux architecture



Introduction and history

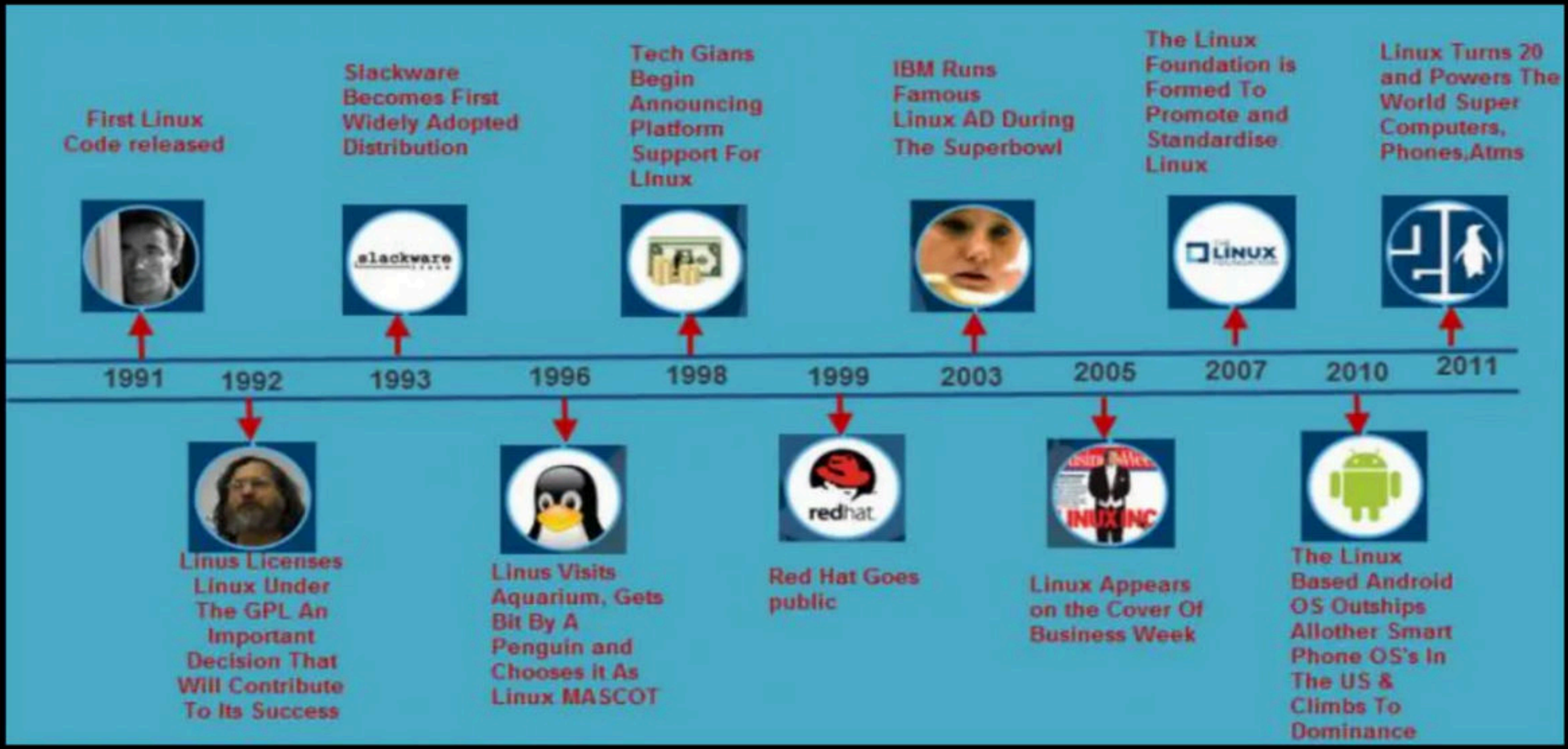
To understand the origins of Linux we need to know about the GNU project as well. The GNU Project started in 1984 with the main aim of developing open source softwares.

Free Software Foundation(FSF) was started in 1985 to raise funds for the GNU project.

Introduction and history

- A Unix-like operating system includes a kernel, compilers, editors, text formatters, mail software, graphical interfaces, libraries, games and many other things.
- By 1990 as part of the GNU project they had developed all components required for a Unix-like OS except the kernel.
- So when Linus Torvalds developed the Linux kernel in 1991 and made it free software in 1992 the combined version of the GNU software and Linux kernel became GNU/Linux systems which are later distributed with different additional features.
- Commonly they are called Linux operating systems even though Linux is only a kernel but not an OS.

The history and timeline of Linux over the years:

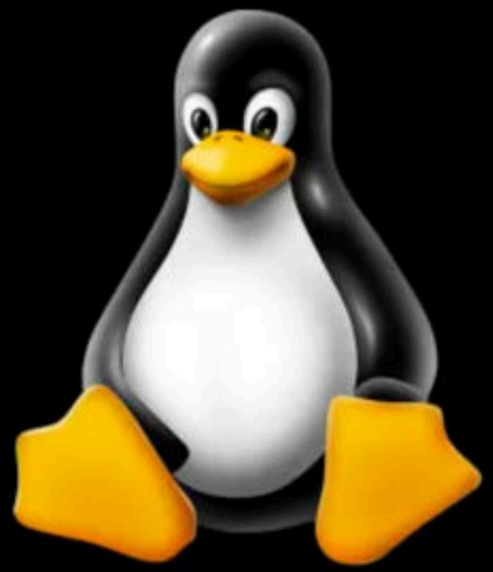


VS

Yes

Open source?

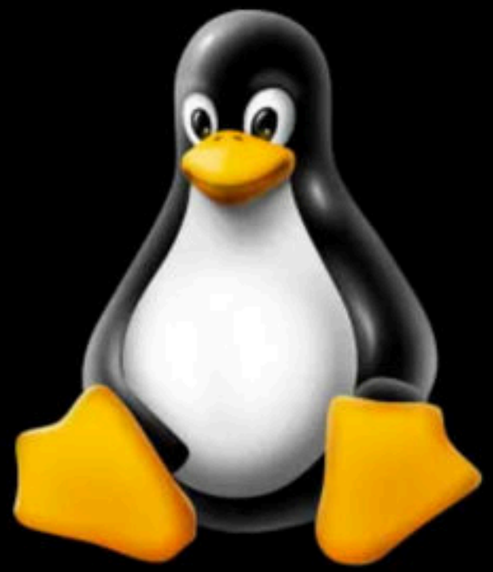
No



Yes

Customizable?

No

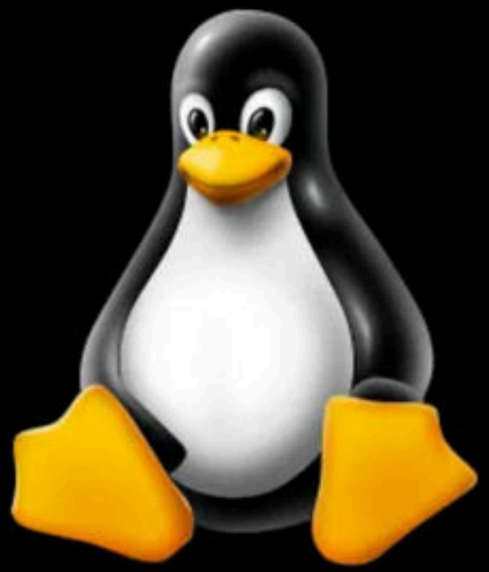


Highly secure

Security

Vulnerable to viruses
and malware attacks.

File names



Case sensitive
(Sample, sample can be given for two file names in the same folder)

Case insensitive
(Sample, sample will be considered as the same name)

File systems supported

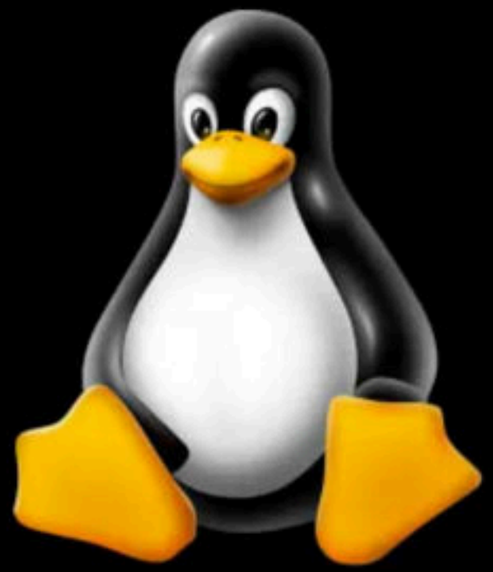
EXT2, EXT3, EXT4, Reisers FS,
XFS and JFS

FAT, FAT32, NTFS
and ReFS

Type of kernel used

Monolithic kernel which consumes more running space

Micro kernel which takes less space but system running efficiency is lower than Linux



I/O devices

Linux considers all the I/O devices like printers, CD-ROMs, and hard drives as files

Windows treats the I/O devices as devices only.

To conclude:

- Windows is simple to use but not open source OS, whereas Linux is open source, customizable and secure but kind of complex for the users having no programming background.

To conclude:

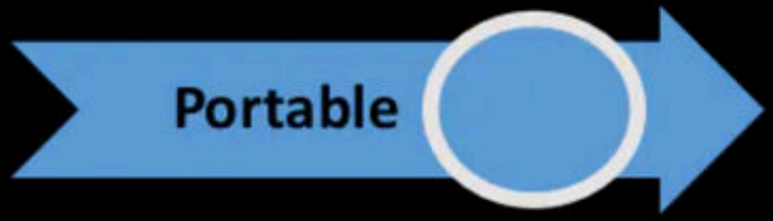
- Linux is more reliable than windows.

To conclude:

- If you are good at programming you can download the Linux source code and modify it according to your requirements and distribute it either commercially or non-commercially. It is that simple.

Following are some of Linux distributions that are commonly used:

1. Debian
2. Ubuntu
3. Linux Mint
4. Gentoo
5. Red Hat Enterprise Linux - designed for commercial purposes
6. CentOS
7. Fedora
8. Arch Linux - designed for advanced or Linux experts
9. OpenSUSE



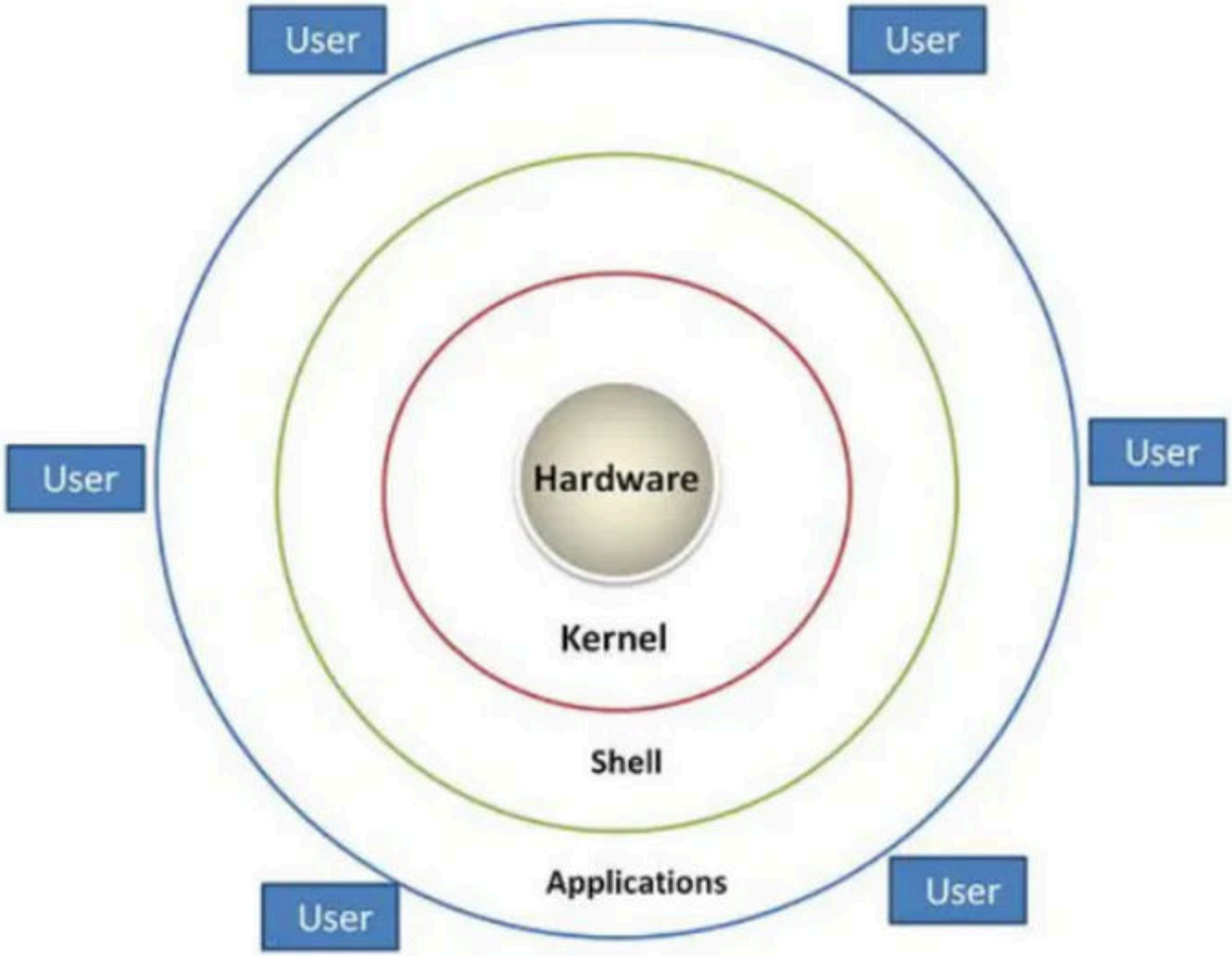
Linux operating system can work on different types of hardware as well as Linux kernel supports the installation of any kind of hardware platform.

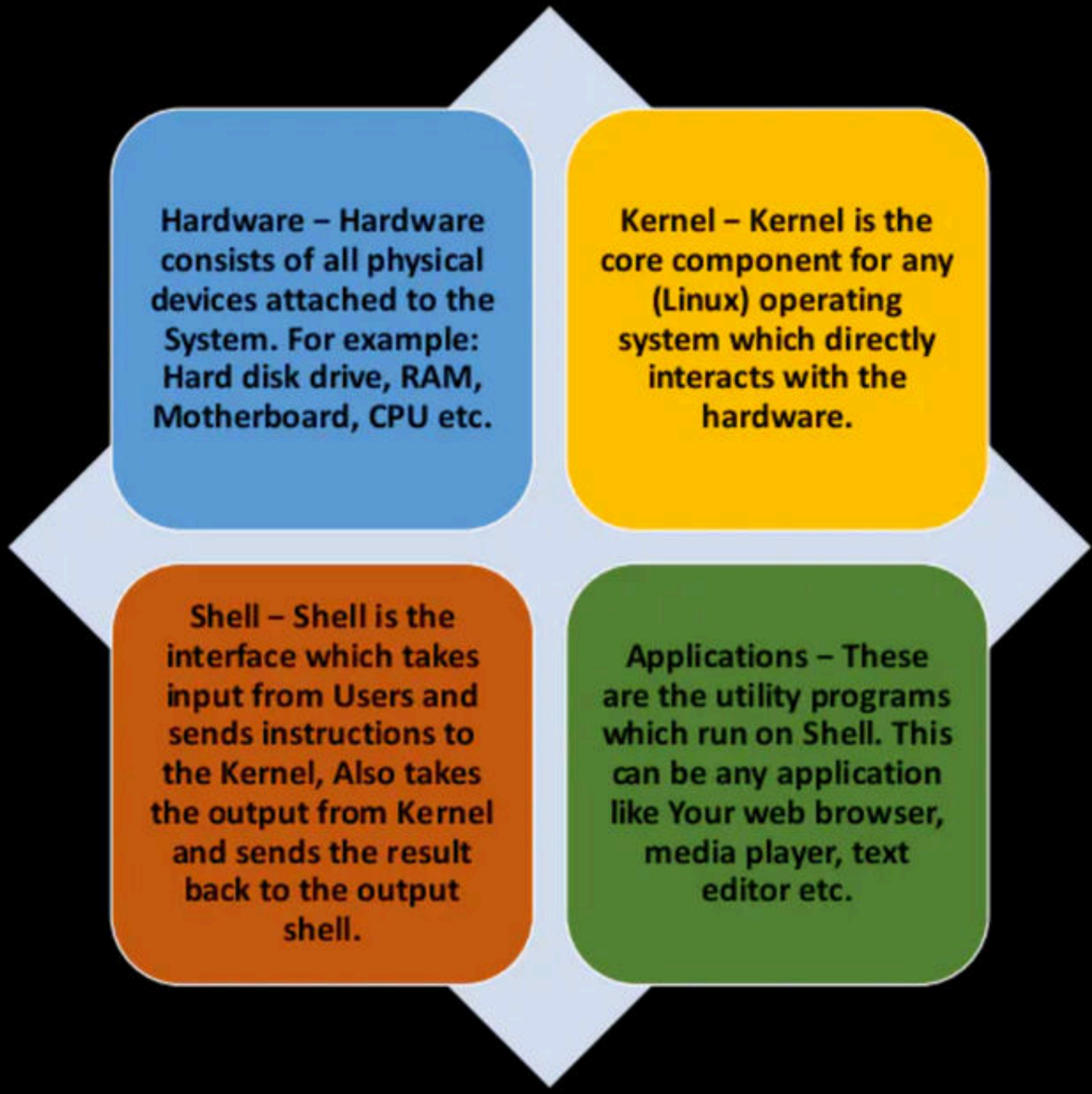
Linux operating system is a multiuser system, which means, multiple users can access the system resources like RAM, Memory, or Application programs at the same time.

Linux operating system affords a standard file structure in which system files or user files are arranged in a hierarchical manner.

Linux operating system offers a special interpreter program called shell, that can be used to execute commands of the OS. It can be used to do several types of operations like call application programs, and so on.

1.3 Linux Architecture




Hardware – Hardware consists of all physical devices attached to the System. For example: Hard disk drive, RAM, Motherboard, CPU etc.
Kernel – Kernel is the core component for any (Linux) operating system which directly interacts with the hardware.
Shell – Shell is the interface which takes input from Users and sends instructions to the Kernel, Also takes the output from Kernel and sends the result back to the output shell.
Applications – These are the utility programs which run on Shell. This can be any application like Your web browser, media player, text editor etc.

All the Linux commands are run in the **terminal** provided by the system.

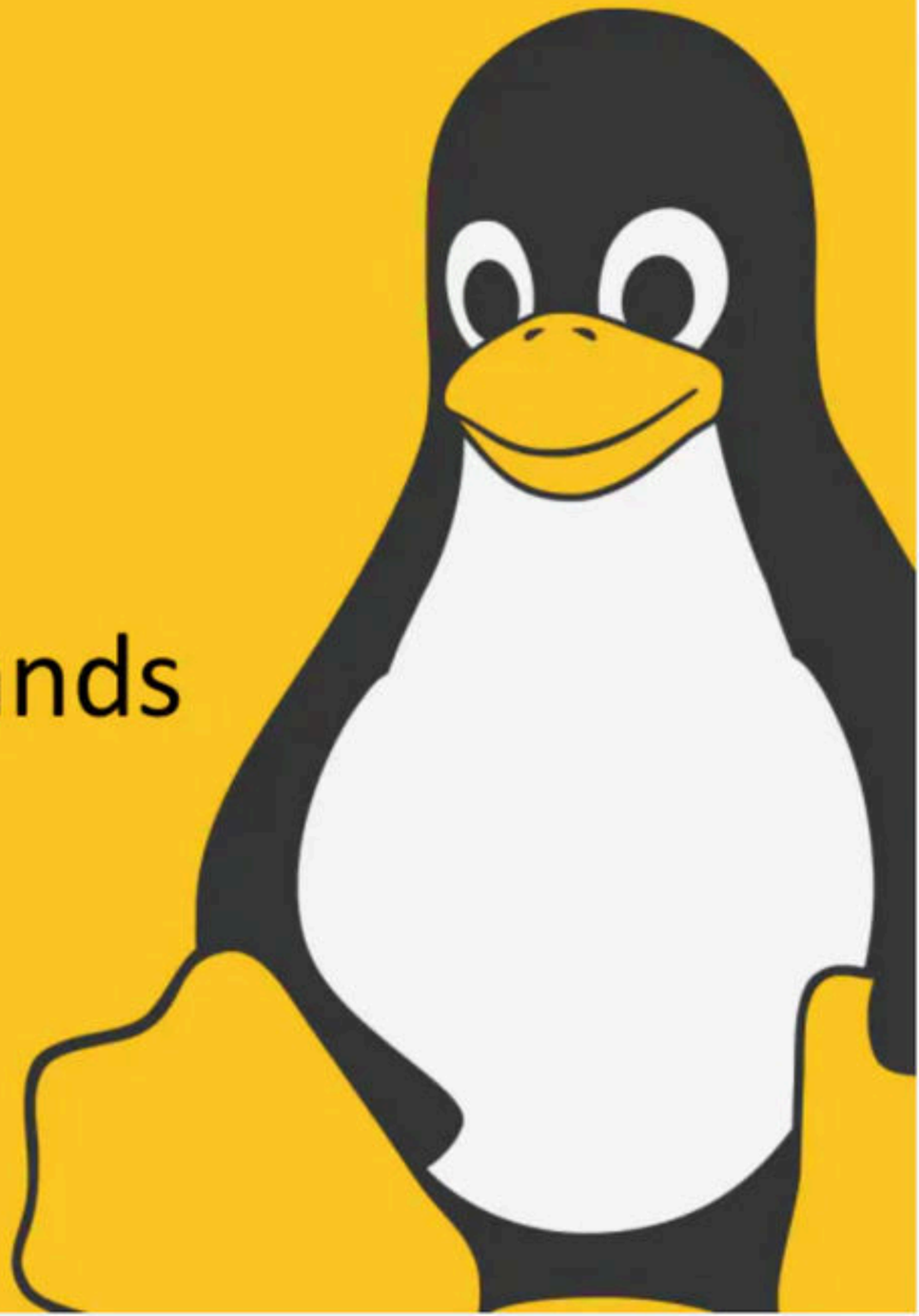
About the terminal:

- This terminal is just like the command prompt of Windows OS.
- The terminal can be used to accomplish all administrative tasks. This includes package installation, file manipulation, and user management.
- We can know the syntax of each command by using **man** command which will print the usage manual for that specific command. For eg. '**man date'** will show the meaning of the command **date** and all the options available with that command.

	Command	Meaning of the command
1	date	Displays the current date and time
2	cal	Shows the current month’s calendar
3	whoami	Displays the current logged in username
4	who	Displays which users are current logged in systems
5	uptime	Displays the current uptime of the system
6	w	Similar to uptime with additional information about the user who logged in
7	groups	Prints the list of groups the current user belong to
8	id	Prints the username and group names and numeric ID’s (UID or group ID) of the current user
9	uname	Displays the information about the system
10	date	Displays the current date and time

LINUX
LECTURE 2

Important Network Commands



THE SEQUENCE OF STEPS IN THE BACKGROUND WHEN WE ENTER A COMMAND:

1

When we open the terminal a parent process will be created and it will keep running in a loop waiting for the user input.

2

Once we give a command say ‘date’ and press ‘Enter’, a child process is created using the system call ‘fork()’.

3

The system will find the location of the executable file for the corresponding command and execute in the child process. Then the shell requests the kernel through system calls to perform the given command. During the execution of the child process the parent process waits for the child to complete.

4

After the child process terminates and returns the output to screen.. the parent process takes over and waits for the next input from the user.


ifconfig

By default **network related command** may not be present in the systems. So by installing '**net-tools**' software we can get the **network related commands**.

To install net tools : **sudo apt-get update**

sudo apt-get install net-tools

sudo apt-get install ifconfig

ifconfig command

Linux ifconfig stands for **interface configurator**.
ifconfig is a command line tool for UNIX-like systems that allows for diagnosing and configuring network interfaces. At boot time, it sets up network interfaces such as Loopback and Ethernet. Most of the time, however, ifconfig is used for network diagnostics.

1.) ifconfig
2.) ifconfig –a
3.) sudo ifconfig enp1s0 up
4.) sudo ifconfig enp1s0 down

Network interface

A network interface is a software interface to networking hardware. Linux kernel distinguishes between two types of network interfaces: physical and virtual.

Physical network interface represents an actual network hardware device such as network interface controller (NIC).
In practice, you'll often find eth0 interface, which represents Ethernet network card.

Virtual network interface doesn't represent any hardware device and is usually linked to one. There are different kinds of virtual interfaces: Loopback, bridges, VLANs, tunnel interfaces and so on. With proliferation of software defined networks, virtual interfaces become wildly used.


ping

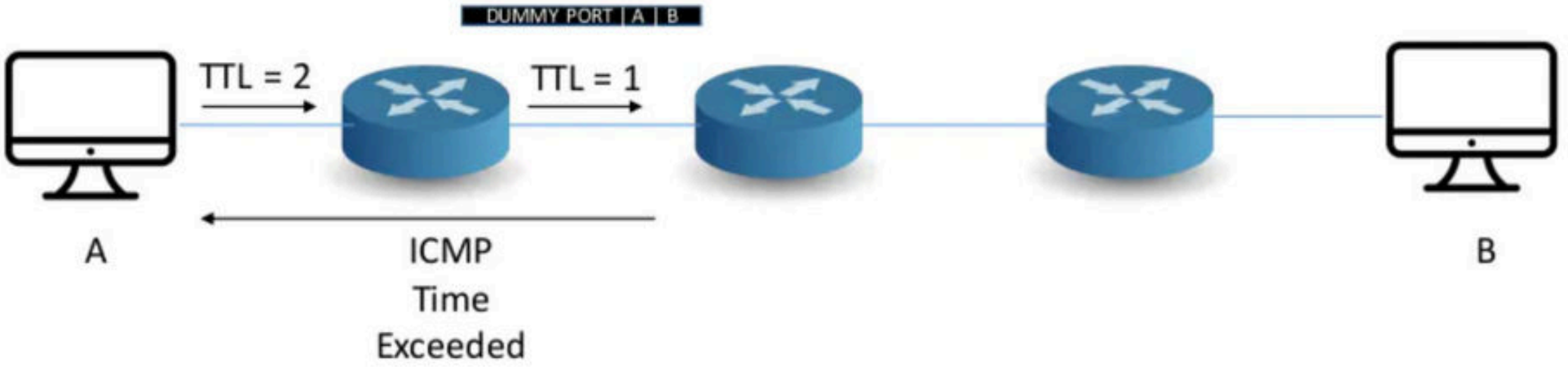
ping command

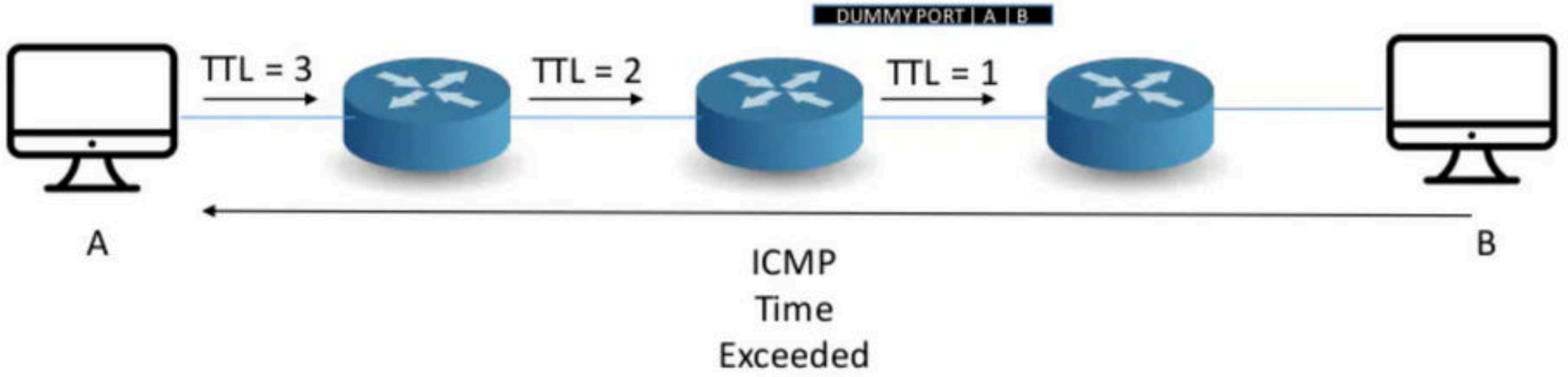
It checks the network connectivity between two nodes.
ping stands for **Packet INternet Groper.**

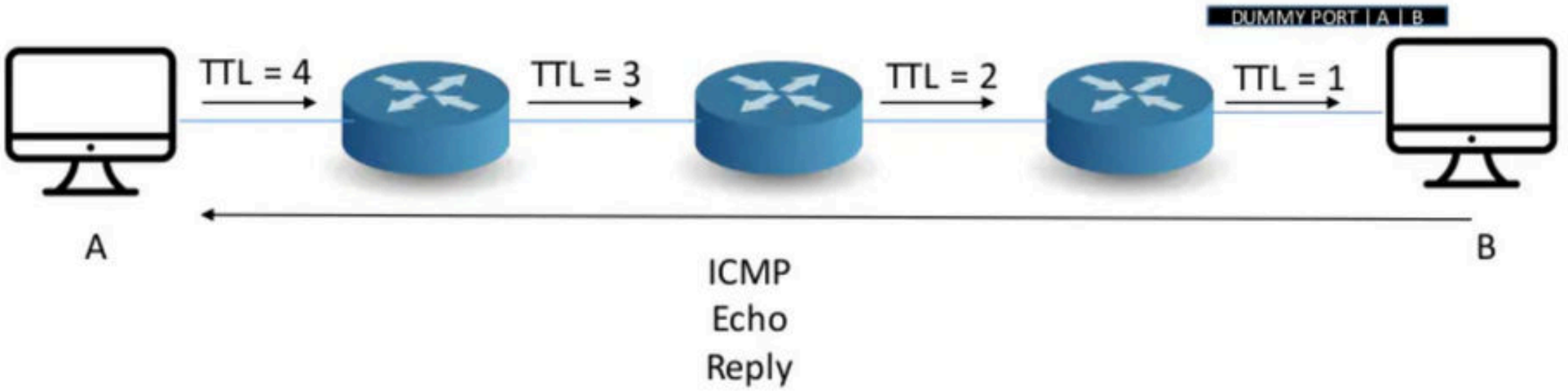
The ping command sends the ICMP echo request to check the network connectivity.
It keeps executing until it is interrupted.
Use Ctrl+C Key to interrupt the execution.

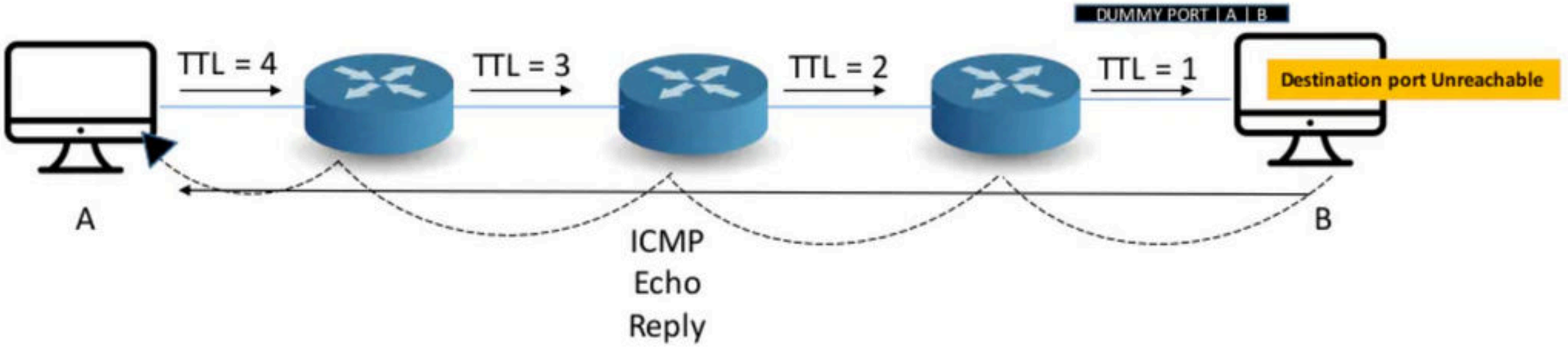
Trace route

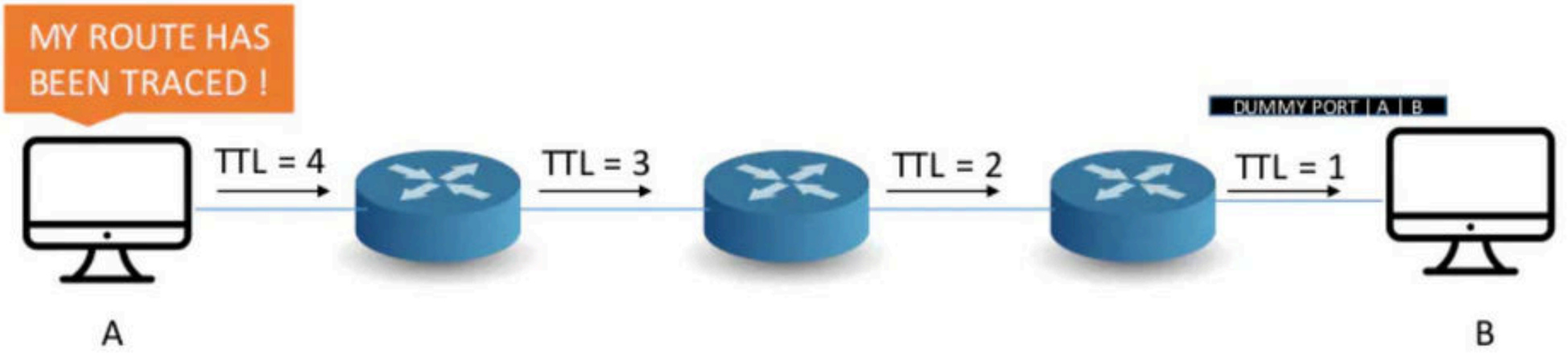
Command on Linux **traceroute** domain


DUMMY PORT | A | B
TTL = 2
TTL = 1
A
ICMP
Time
Exceeded
B


DUMMY PORT | A | B
TTL = 3
TTL = 2
TTL = 1
A
B
ICMP
Time
Exceeded

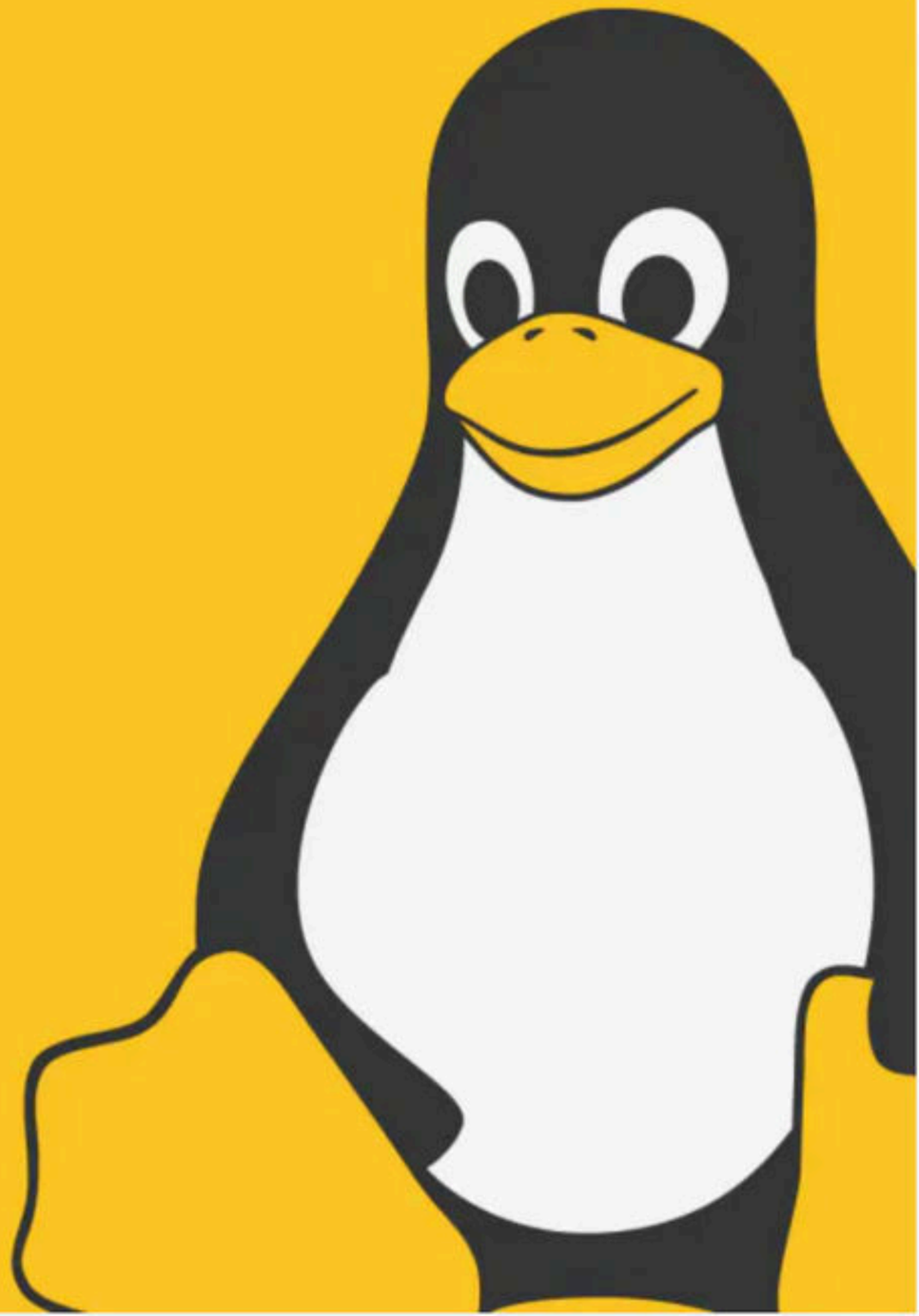

DUMMY PORT | A | B
TTL = 4
TTL = 3
TTL = 2
TTL = 1
A
B
ICMP
Echo
Reply


DUMMY PORT | A | B
TTL = 4
TTL = 3
TTL = 2
TTL = 1
Destination port Unreachable
A
B
ICMP
Echo
Reply


MY ROUTE HAS BEEN TRACED !
TTL = 4
TTL = 3
TTL = 2
TTL = 1
DUMMY PORT | A | B
A
B

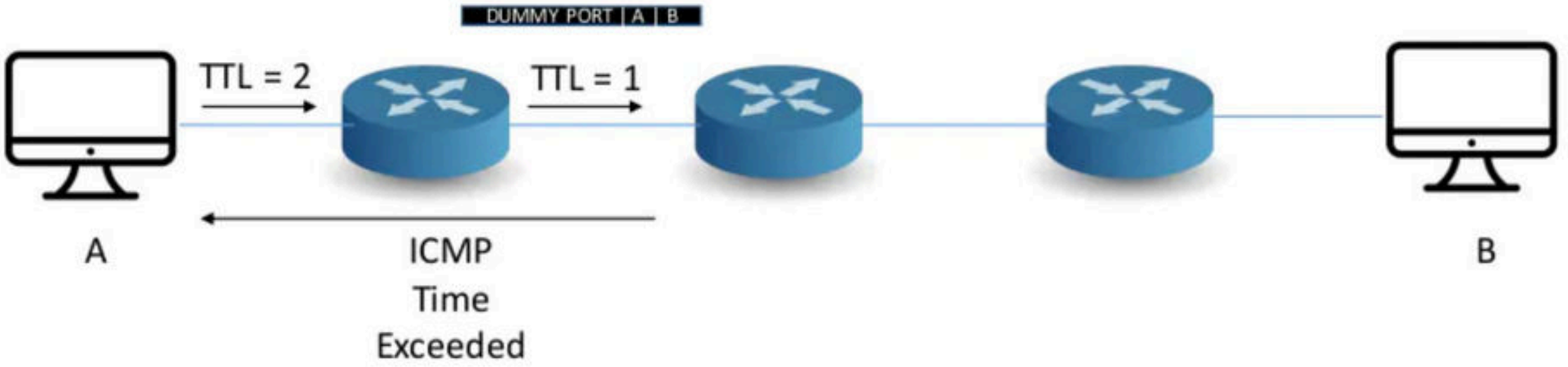
COMMAND	DESCRIPTION
ip	It is a replacement of ifconfig command.
netstat	Display connection information.
ss	It is a replacement of netstat.
dig	Query DNS related information.
nslookup	Find DNS related query.
route	Shows and manipulate IP routing table.
host	Performs DNS lookups.
arp	View or add contents of the kernel's ARP table.
iwconfig	Used to configure wireless network interface.
hostname	To identify a network name.
curl or wget	To download a file from internet.
mtr	Combines ping and tracepath into a single command.
whois	Will tell you about the website's whois.
ifplugstatus	Tells whether a cable is plugged in or not.

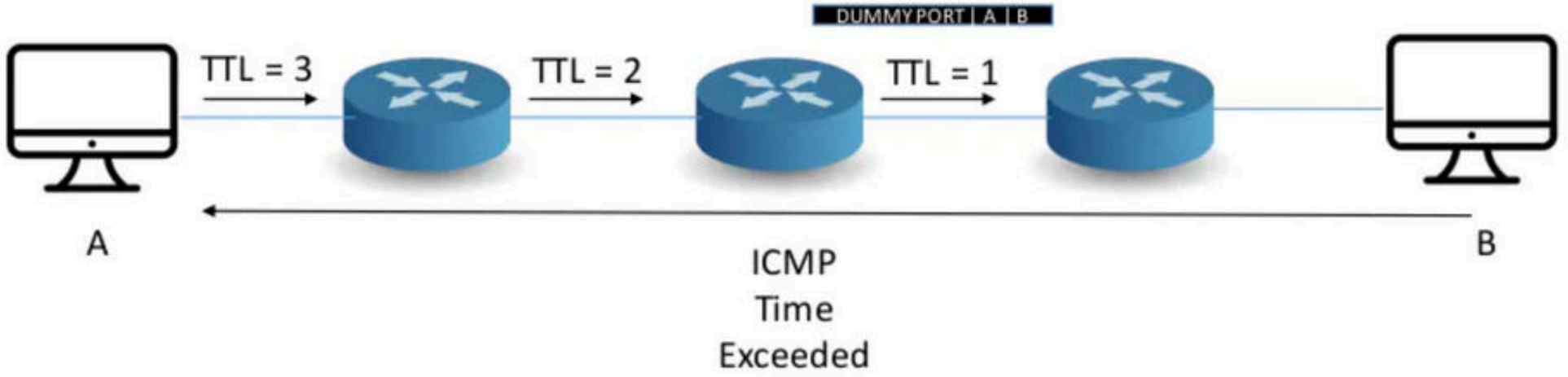
LINUX
LECTURE 3
Network Commands

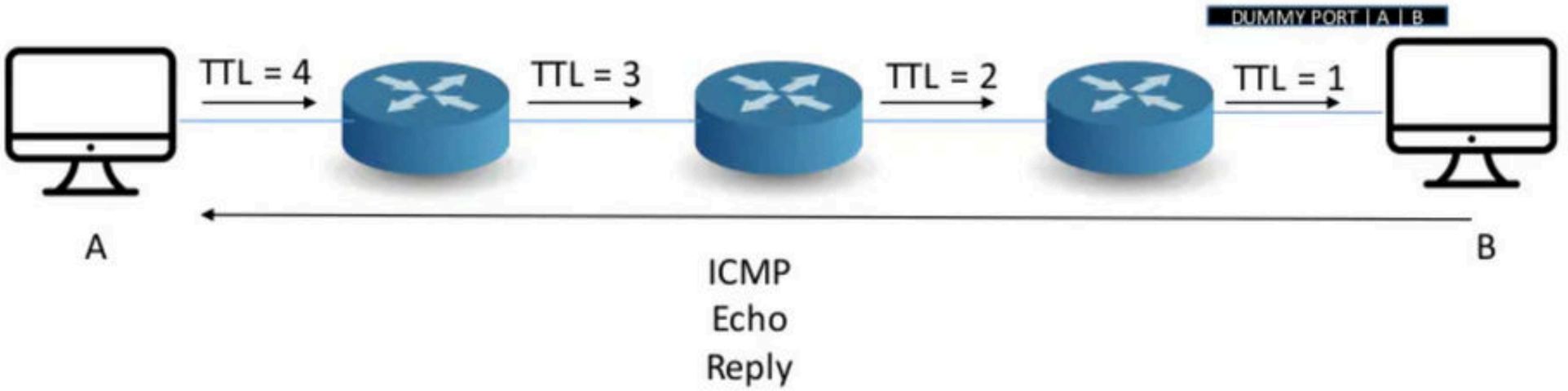


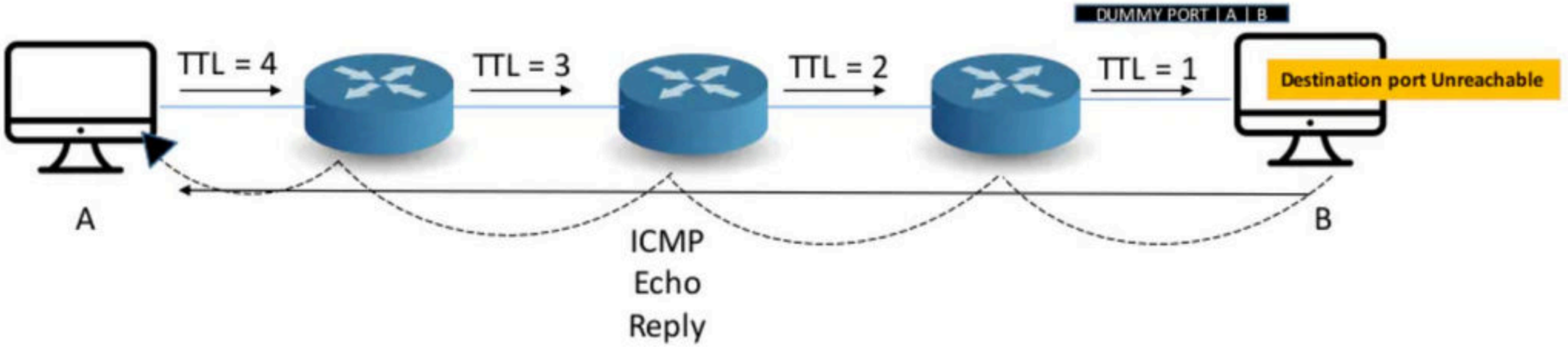
Trace route

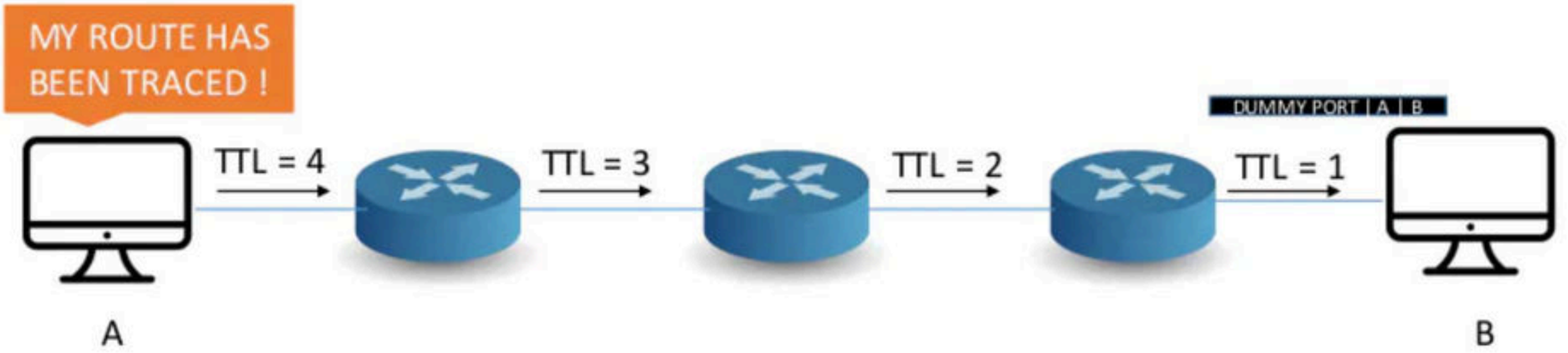
Command on Linux **traceroute** domain


DUMMY PORT | A | B
TTL = 2
TTL = 1
A
ICMP
Time
Exceeded
B


DUMMY PORT | A | B
TTL = 3
TTL = 2
TTL = 1
A
B
ICMP
Time
Exceeded


DUMMY PORT | A | B
TTL = 4
TTL = 3
TTL = 2
TTL = 1
A
B
ICMP
Echo
Reply


DUMMY PORT | A | B
TTL = 4
TTL = 3
TTL = 2
TTL = 1
Destination port Unreachable
A
B
ICMP
Echo
Reply


MY ROUTE HAS BEEN TRACED !
TTL = 4
TTL = 3
TTL = 2
TTL = 1
DUMMY PORT | A | B
A
B

COMMAND	DESCRIPTION
ip	It is a replacement of ifconfig command.
netstat	Display connection information.
ss	It is a replacement of netstat.
dig	Query DNS related information.
nslookup	Find DNS related query.
route	Shows and manipulate IP routing table.
host	Performs DNS lookups.
arp	View or add contents of the kernel's ARP table.
iwconfig	Used to configure wireless network interface.
hostname	To identify a network name.
curl or wget	To download a file from internet.
mtr	Combines ping and tracepath into a single command.
whois	Will tell you about the website's whois.
ifplugstatus	Tells whether a cable is plugged in or not.

What Is Kernel?

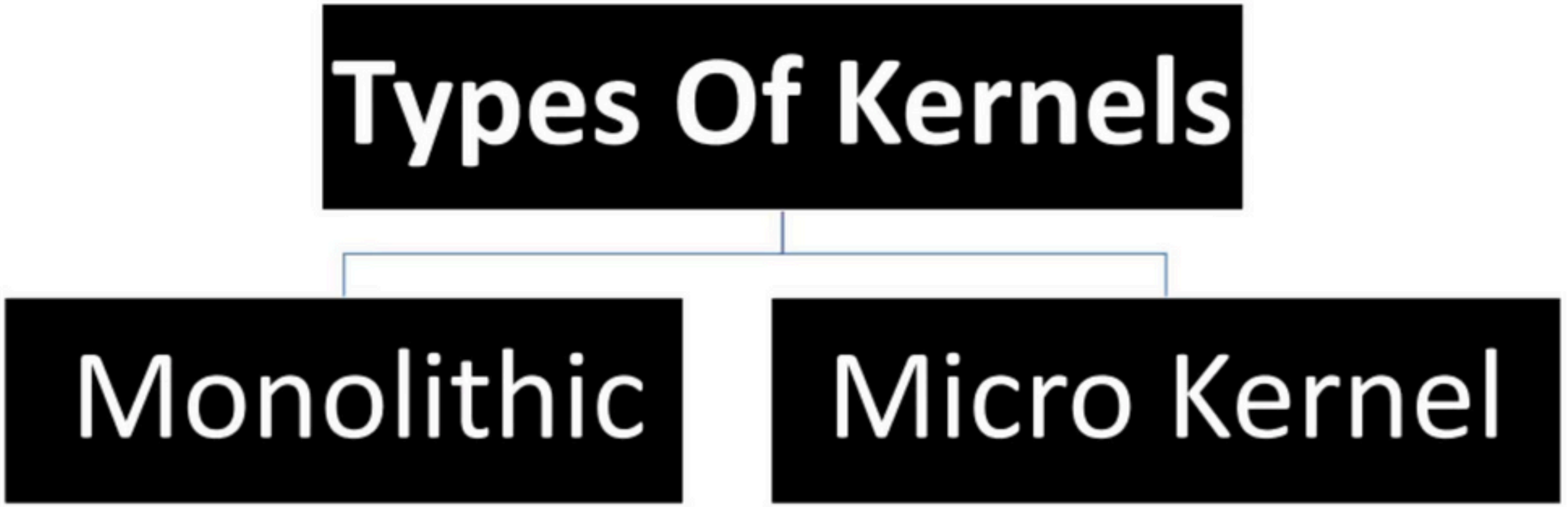
A kernel is a central component of an operating system.

It acts as an interface between the user applications and the hardware.

The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc).

The main tasks of the kernel are

- Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system management


Types Of Kernels
Monolithic
Micro Kernel

Monolithic Kernels

Earlier in this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc were packaged into a single module in the memory address space allocated for the kernel.

Advantage

Process executions are faster as the whole kernel is in the single memory address space and there is no need to switch between user mode and kernel mode for all of the system services.

Disadvantages

- Big kernel size
- Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel that can take hours.

- **Later the monolithic kernel is modularized** in such a way that it contains modules for different services which can be loaded and unloaded.
- Maintainability of the kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module.
- So, there is no need to bring down and recompile the whole kernel for a smallest bit of change.
- **Linux follows the monolithic modular approach**.

Microkernels

- Basic services like device driver management, protocol stack, file system etc are allocated memory in user space which is a separated memory address space other than the kernel space.
- This reduces the kernel code size as we have the bare minimum code running in the kernel.
- So, if a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

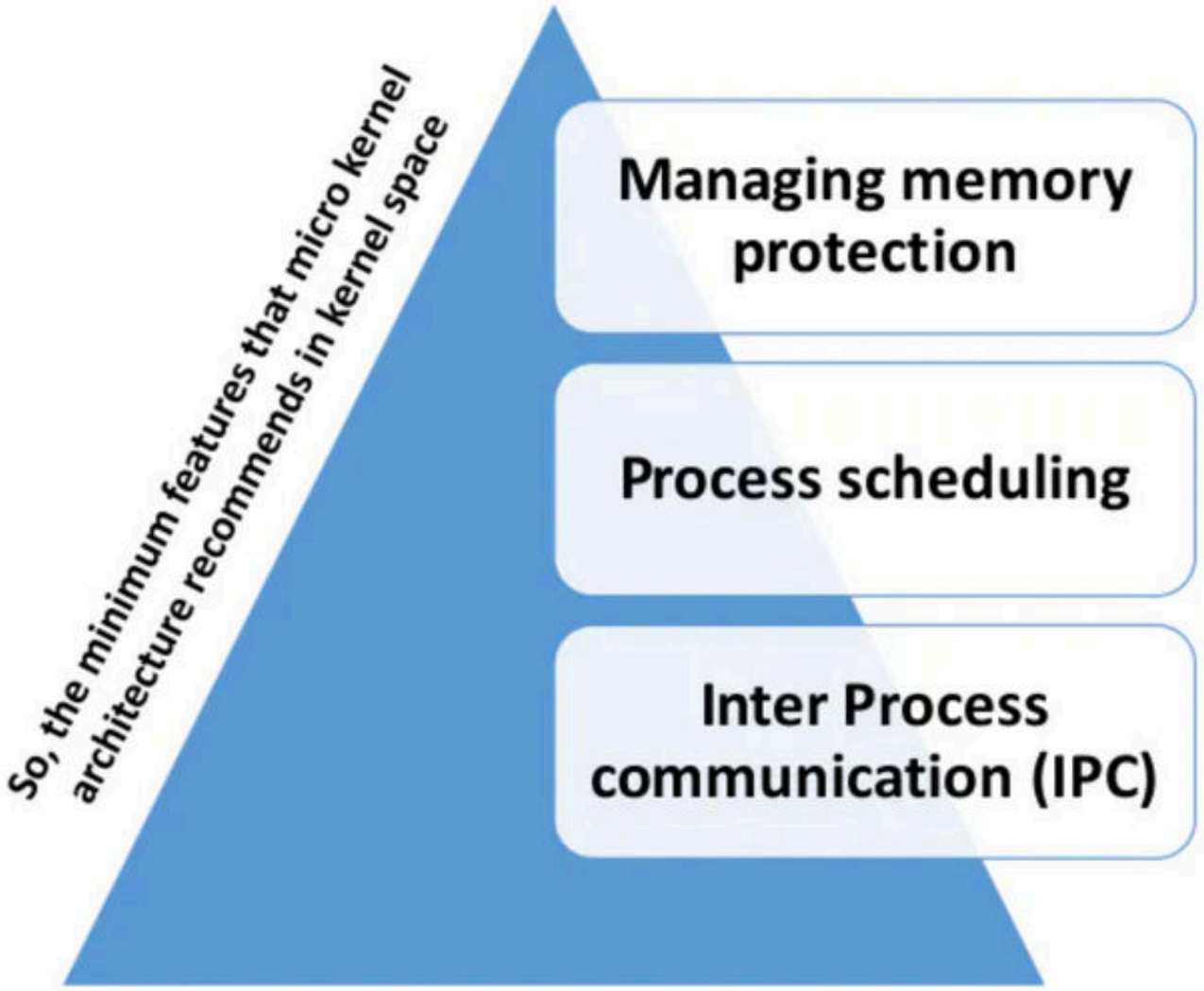
Microkernels

All the basic OS services which are made part of user space are made to run as servers and other programs in the system communicate with them through **inter process communication (IPC)**.

eg: there are servers for device drivers, network protocol stacks, file systems, graphics, etc.

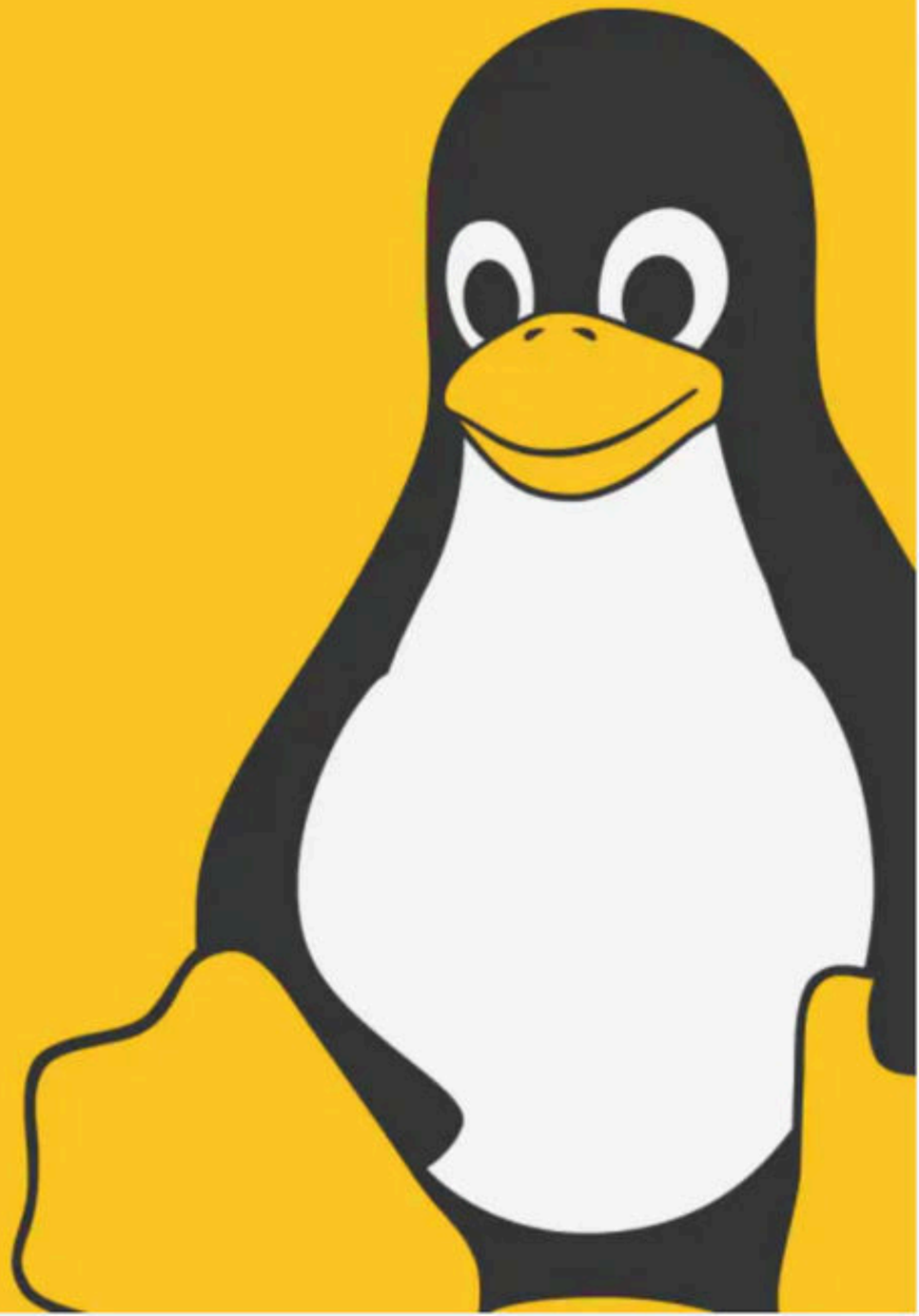
Microkernels

- Microkernel servers are essentially **daemon programs(which run in the background)** like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs.
- This allows some servers, particularly device drivers, to interact directly with hardware.
- These servers are started at the system start-up.

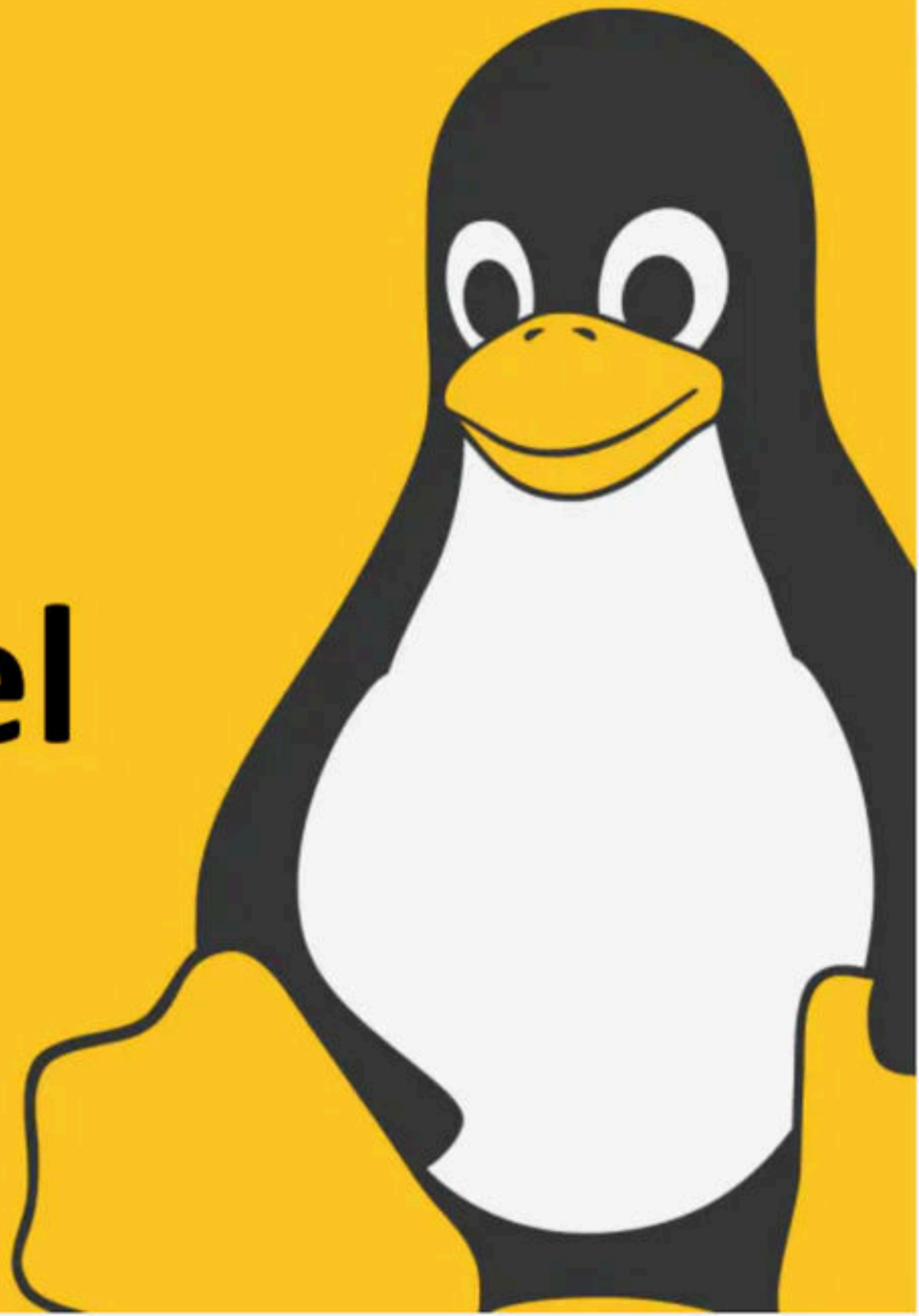

So, the minimum features that micro kernel architecture recommends in kernel space
Managing memory protection
Process scheduling
Inter Process communication (IPC)

LINUX
LECTURE 4

Types of Kernel
And File system



Kernel
and
Types of Kernel



What Is Kernel?

A kernel is a central component of an operating system.

It acts as an interface between the user applications and the hardware.

The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc).

The main tasks of the kernel are

- Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system management

Types Of Kernels

- Monolithic
- Micro Kernel

Monolithic Kernels

Earlier in this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc were packaged into a single module in the memory address space allocated for the kernel.

Advantage

Process executions are faster as the whole kernel is in the single memory address space and there is no need to switch between user mode and kernel mode for all of the system services.

Disadvantages

Big kernel size

Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel that can take hours.

- **Later the monolithic kernel is modularized** in such a way that it contains modules for different services which can be loaded and unloaded.
- Maintainability of the kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module.
- So, there is no need to bring down and recompile the whole kernel for a smallest bit of change.
- **Linux follows the monolithic modular approach**.

Microkernels

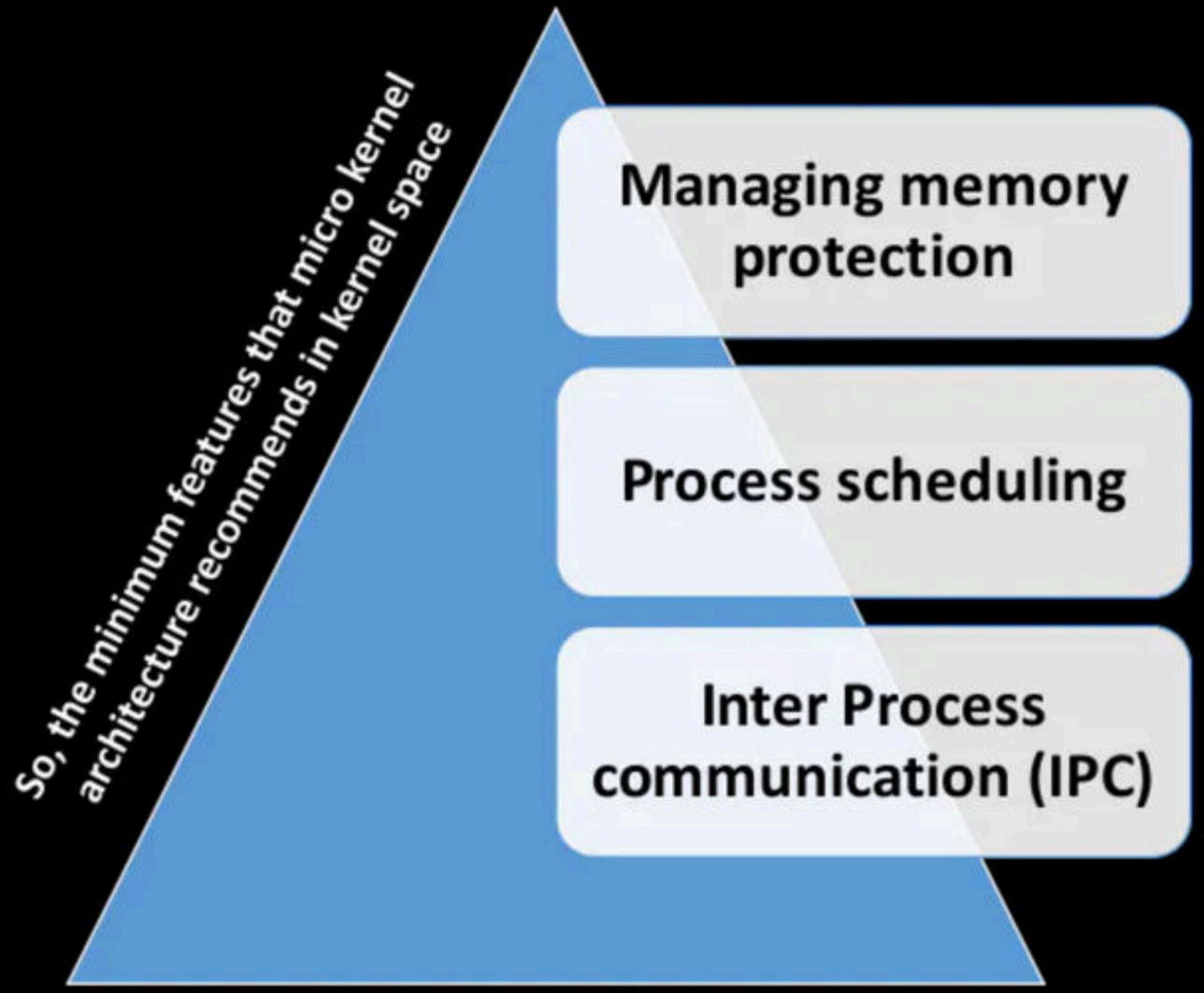
- Basic services like device driver management, protocol stack, file system etc are allocated memory in user space which is a separated memory address space other than the kernel space.
- This reduces the kernel code size as we have the bare minimum code running in the kernel.
- So, if a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

Microkernels

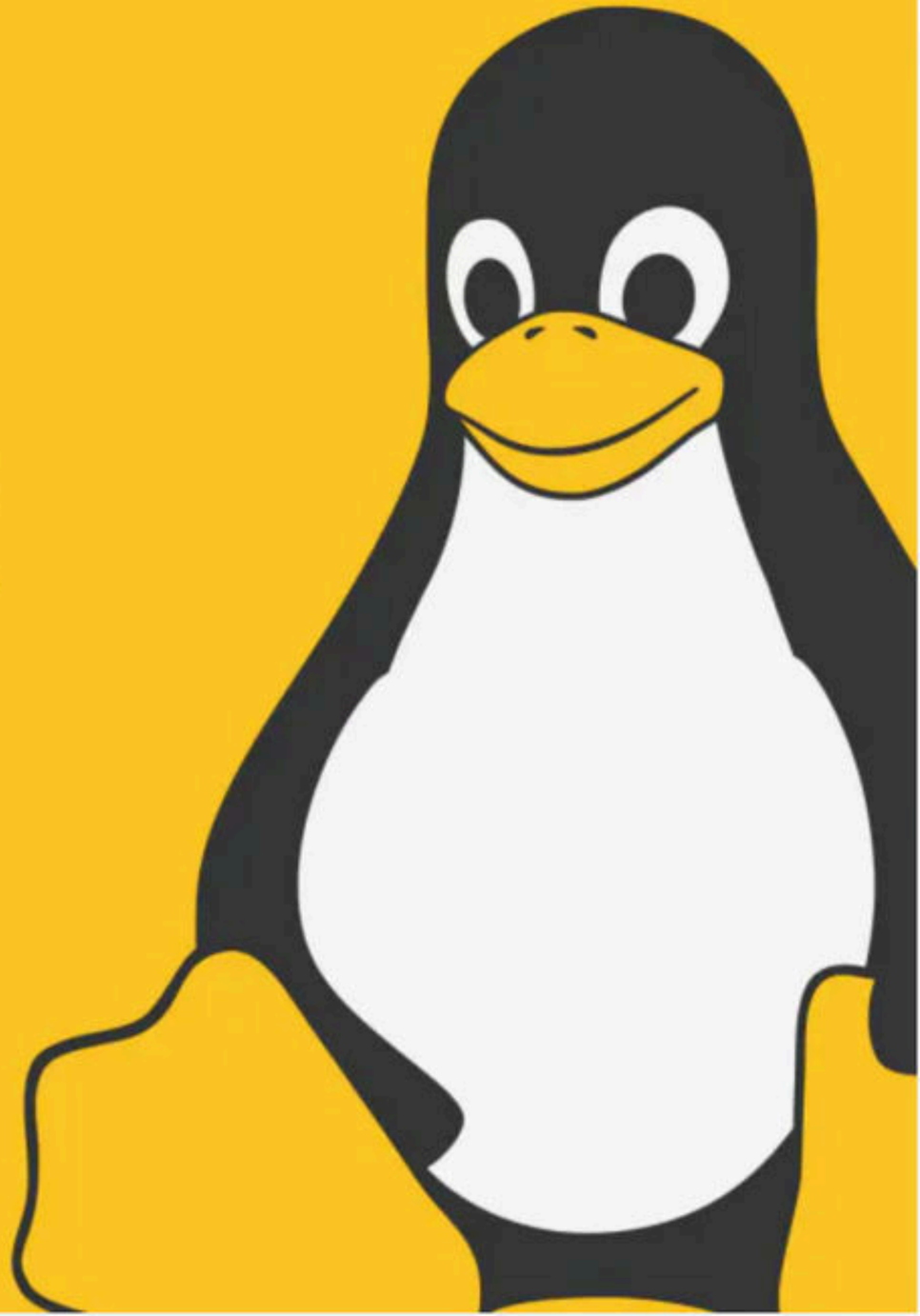
All the basic OS services which are made part of user space are made to run as servers and other programs in the system communicate with them through inter process communication (IPC).
eg: there are servers for device drivers, network protocol stacks, file systems, graphics, etc.

Microkernels

- Microkernel servers are essentially **daemon programs(which run in the background)** like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs.
- This allows some servers, particularly device drivers, to interact directly with hardware.
- These servers are started at the system start-up.

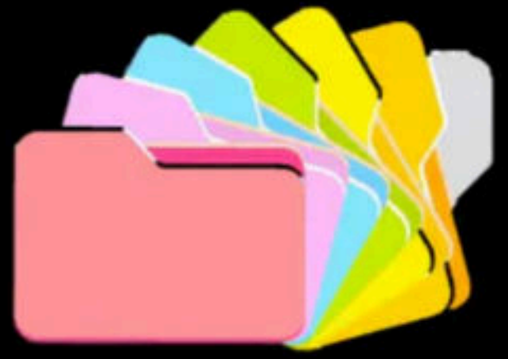

So, the minimum features that micro kernel architecture recommends in kernel space
Managing memory protection
Process scheduling
Inter Process communication (IPC)

Linux File System



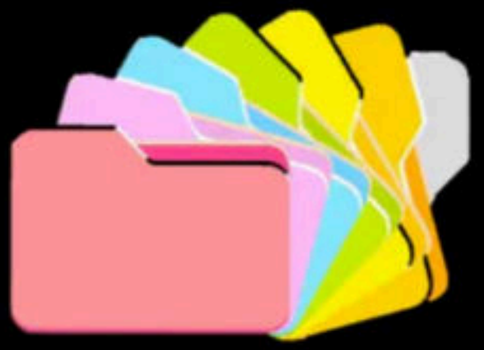
A simple description of the UNIX system, also applicable to Linux, is this:
"On a UNIX system, everything is a file; if something is not a file, it is a process."

Linux (just like UNIX) looks at everything as a file. If you write a program, you add one or more files to the system.

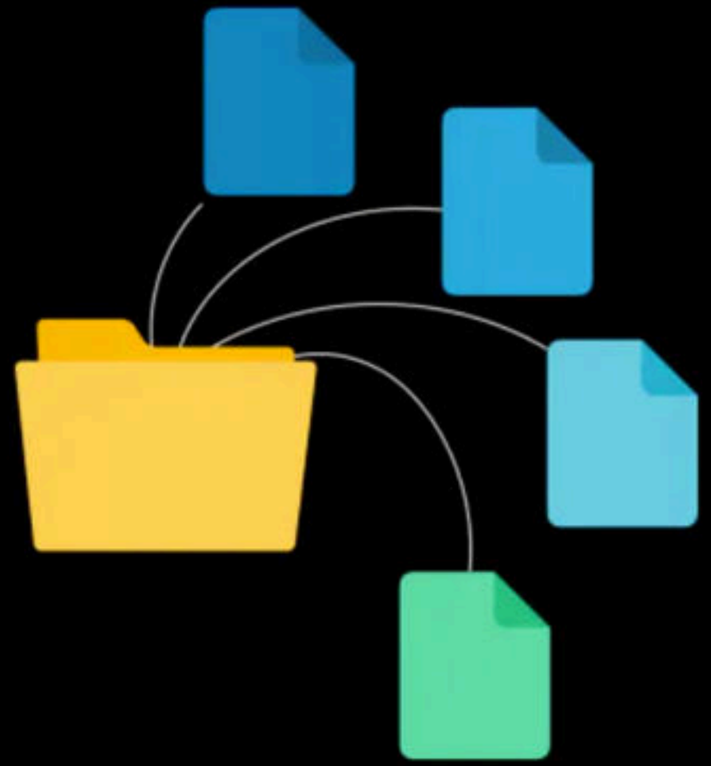


There are special files that are more than just files (named pipes and sockets, for instance), but we can keep things simple, saying that everything is a file.

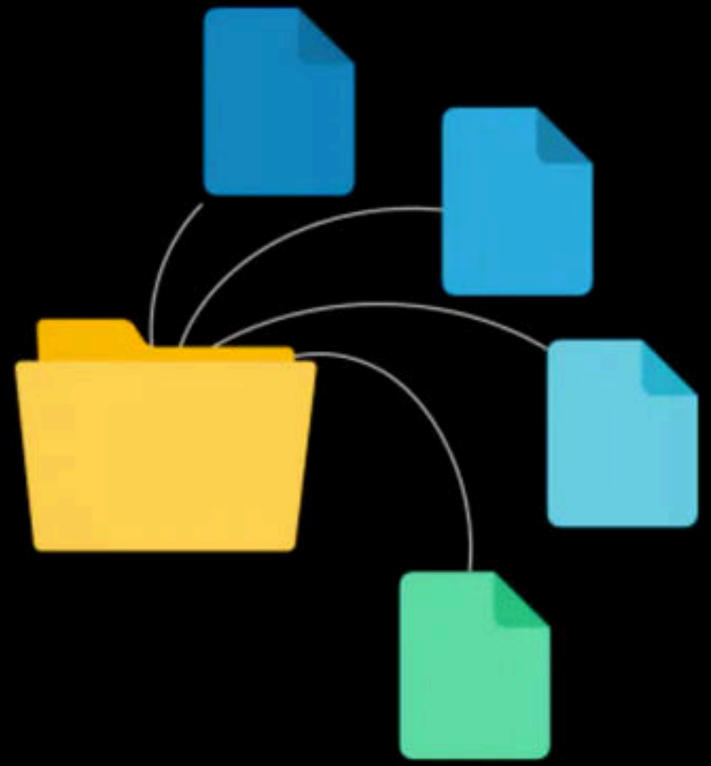
A Linux system, just like UNIX, makes no difference between a file and a directory, since a **directory is just a file containing names of other files.**
In other words, directory is the collection of files and subdirectories.



Directory is also a file and it contains details of the files and subdirectories that it contains.

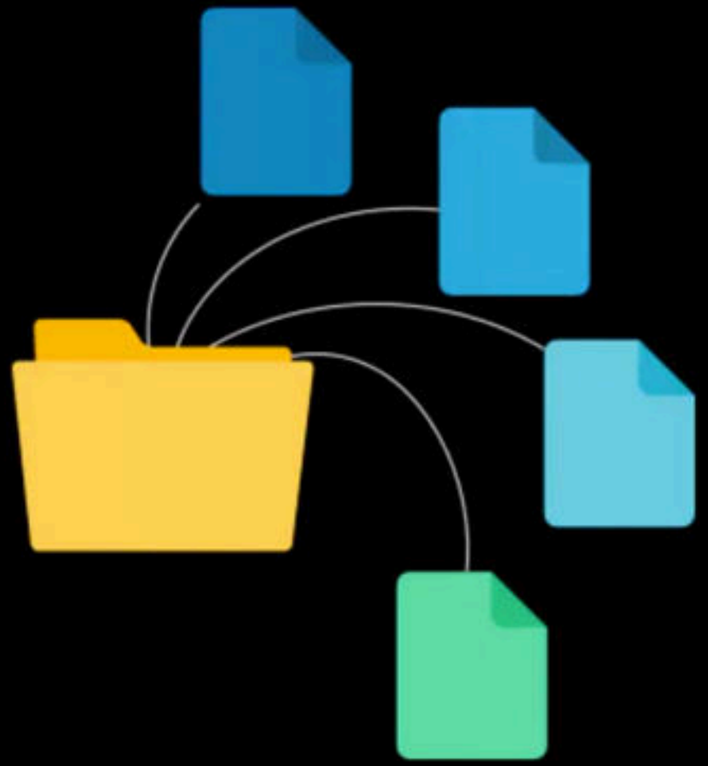


A file has many components like name, contents, administrative info like permissions and modification times etc. **The administrative info related to the file is stored in a data structure called i-node or inode**.

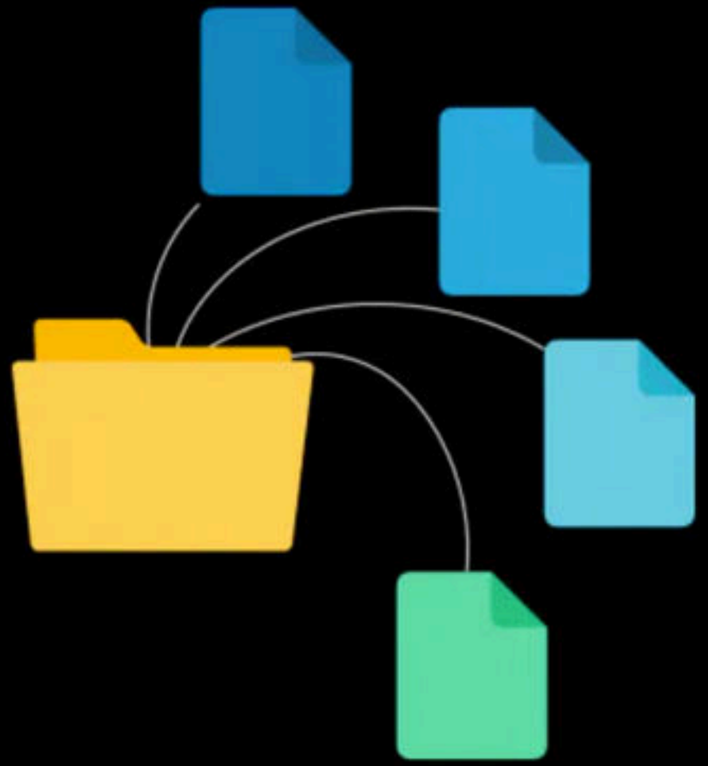


Some more details like file size, the location on the disk where the file contents are stored are also kept in inode. When a new file is created **a unique inode number** is assigned by the OS which is used for accessing and operations on the file throughout the time it exists. So an inode stores the metadata about a file.

The **list of all inodes** in the file system is called **inode table**.



Superblock stores the metadata about the whole filesystem. It has info such as file system type, size, number of empty and filled blocks, size and location of the inode table etc. Superblock is very important for the file system that's why several redundant copies of the super block are stored in different places so that if the primary superblock is corrupted then it can be restored from the backup copies.



Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files.

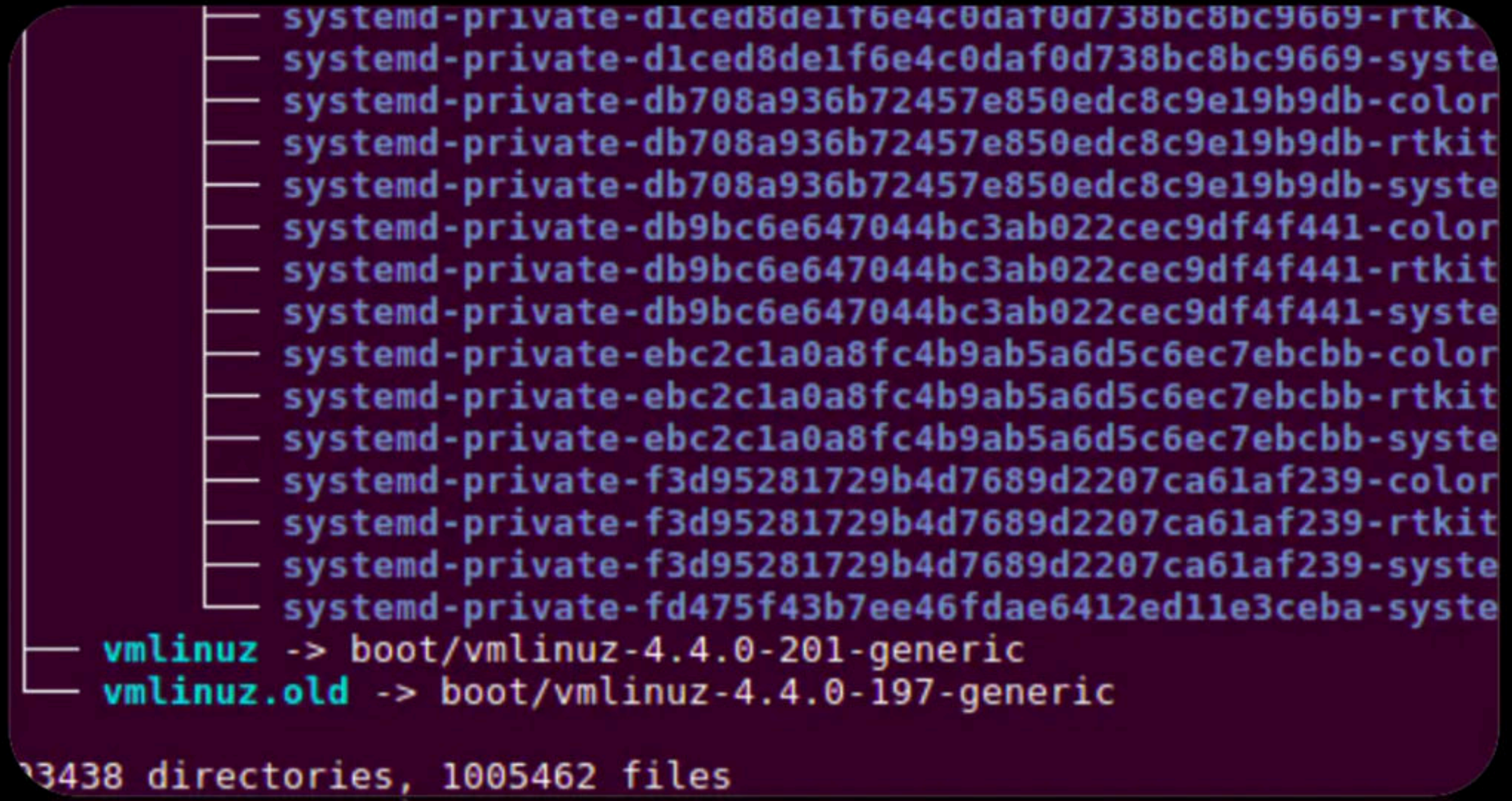
tree command

$sudo apt install tree

$ tree \

“\” represents root

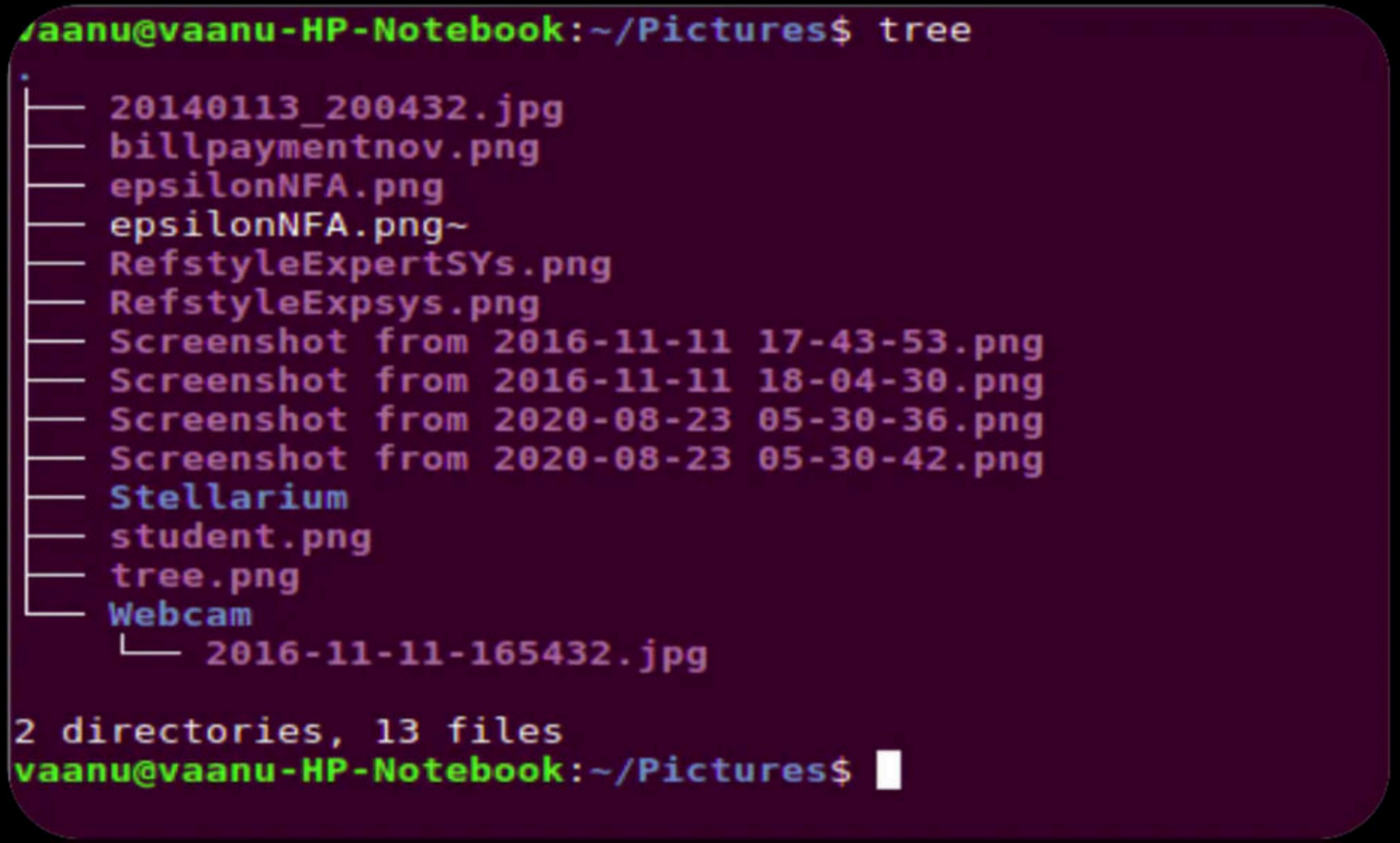
It displays all the folders and file in a tree form


systemd-private-d1ced8de1f6e4c0daf0d738bc8bc9669-rtkit
systemd-private-d1ced8de1f6e4c0daf0d738bc8bc9669-syste
systemd-private-db708a936b72457e850edc8c9e19b9db-color
systemd-private-db708a936b72457e850edc8c9e19b9db-rtkit
systemd-private-db708a936b72457e850edc8c9e19b9db-syste
systemd-private-db9bc6e647044bc3ab022cec9df4f441-color
systemd-private-db9bc6e647044bc3ab022cec9df4f441-rtkit
systemd-private-db9bc6e647044bc3ab022cec9df4f441-syste
systemd-private-ebc2c1a0a8fc4b9ab5a6d5c6ec7ebcbb-color
systemd-private-ebc2c1a0a8fc4b9ab5a6d5c6ec7ebcbb-rtkit
systemd-private-ebc2c1a0a8fc4b9ab5a6d5c6ec7ebcbb-syste
systemd-private-f3d95281729b4d7689d2207ca61af239-color
systemd-private-f3d95281729b4d7689d2207ca61af239-rtkit
systemd-private-f3d95281729b4d7689d2207ca61af239-syste
systemd-private-fd475f43b7ee46fdae6412ed11e3ceba-syste
vmlinuz -> boot/vmlinuz-4.4.0-201-generic
vmlinuz.old -> boot/vmlinuz-4.4.0-197-generic
3438 directories, 1005462 files

Also give total directories and files in your system

If you open terminal in some directory

For ex: I opened terminal in ‘Pictures’ directory (folder)
And run the command
$ tree
It displays all the files and directories in that folder
stellarium and Webcam are directories and rest are files (image files)


vaanu@vaanu-HP-Notebook:~/Pictures$ tree
.
├── 20140113_200432.jpg
├── billpaymentnov.png
├── epsilonNFA.png
├── epsilonNFA.png~
├── RefstyleExpertSYs.png
├── RefstyleExpsys.png
├── Screenshot from 2016-11-11 17-43-53.png
├── Screenshot from 2016-11-11 18-04-30.png
├── Screenshot from 2020-08-23 05-30-36.png
├── Screenshot from 2020-08-23 05-30-42.png
├── Stellarium
├── student.png
├── tree.png
└── Webcam
    └── 2016-11-11-165432.jpg
2 directories, 13 files
vaanu@vaanu-HP-Notebook:~/Pictures$

Most files are just files, called regular files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
While it is reasonably safe to suppose that everything you encounter on a Linux system is a file.

Ordinary file: It is of two types, text file (contains printable characters and we can understand the data in this). Binary files (contains both printable and unprintable characters). Picture, sound, video are binary files.

Directories: files that are lists of other files.

Special files: the mechanism used for input and output. Most special files are in /dev. **This is also known as device file.**

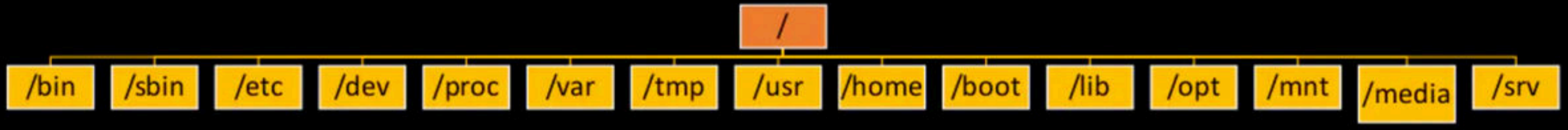
The Parent child relationship

All files in linux are related to one another. The file system in LINUX is a collection of all of these related files (ordinary, directory, device files) organized in a hierarchical (an inverted tree) structure.

There is a top (known as root) which serves as reference point for all files.

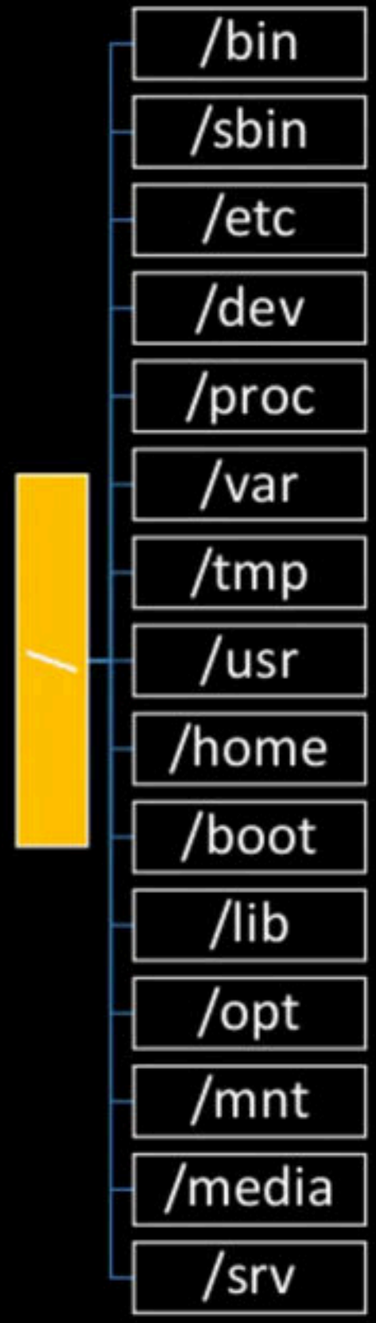
It is represented by -> / (frontslash)

root is actually a directory.



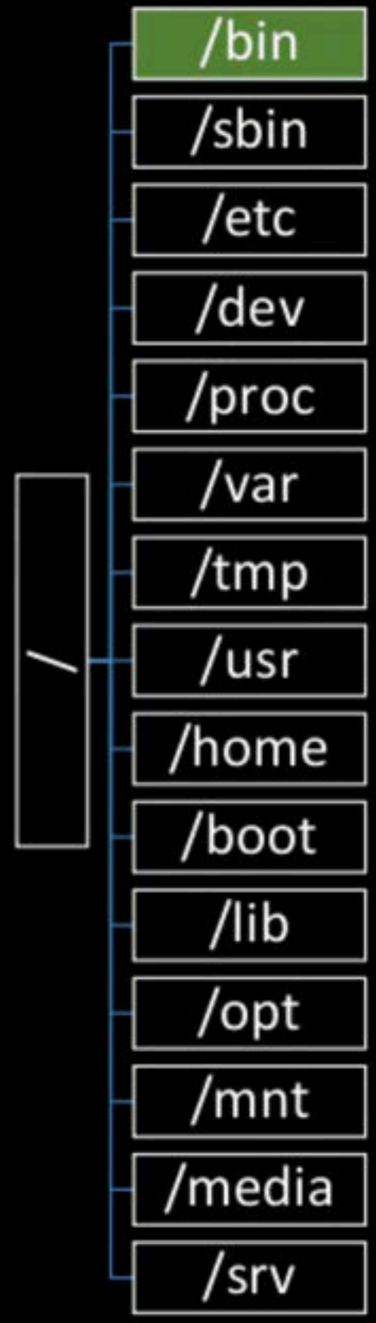
1. / – Root

- Every single file and directory starts from the root directory.
- Only the root user has write privilege under this directory.
- Please note that /root is the root user's home directory, which is not the same as /.



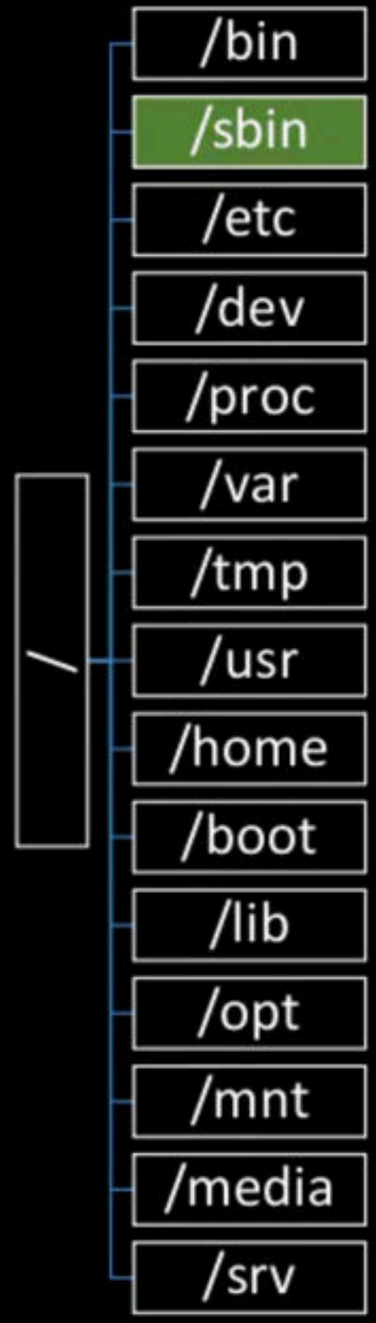
2. /bin – User Binaries

- Contains binary executables.
- Common linux commands you need to use in single-user modes are located under this directory.
- Commands used by all the users of the system are located here.
- For example: ps, ls, ping, grep, cp.



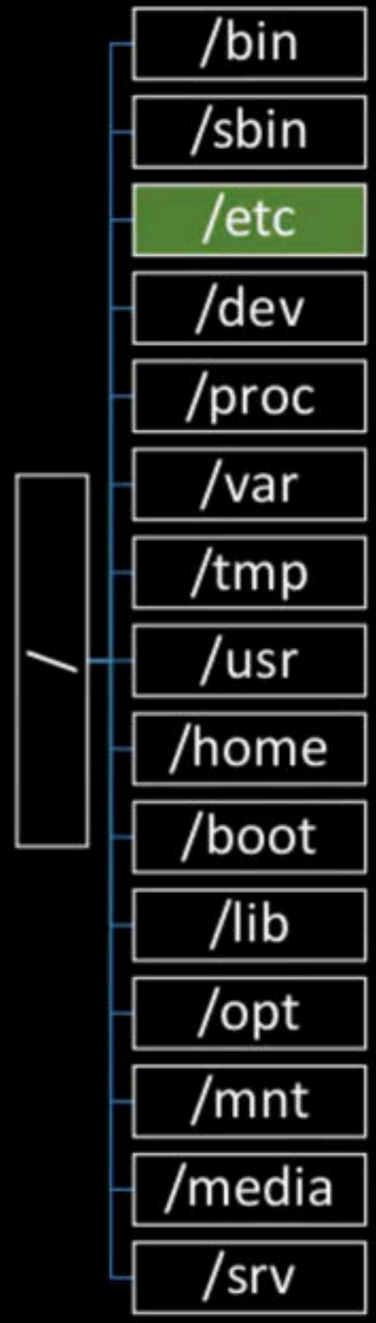
3. /sbin – System Binaries

- Just like /bin, /sbin also contains binary executables.
- But, the linux commands located under this directory are used typically by system administrators, for system maintenance purposes.
- For example: iptables, reboot, fdisk, ifconfig, swapon



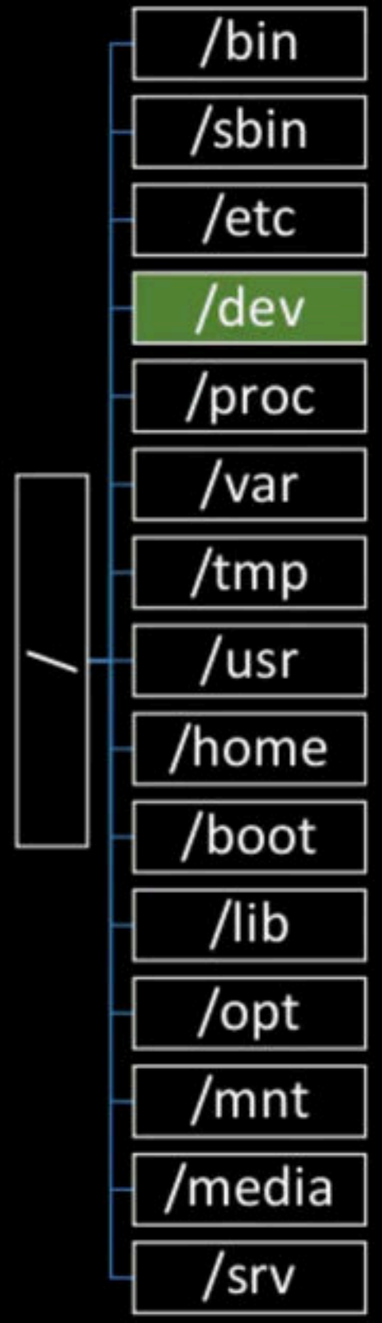
4. /etc – Configuration Files

- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- For example: /etc/resolv.conf, /etc/logrotate.conf



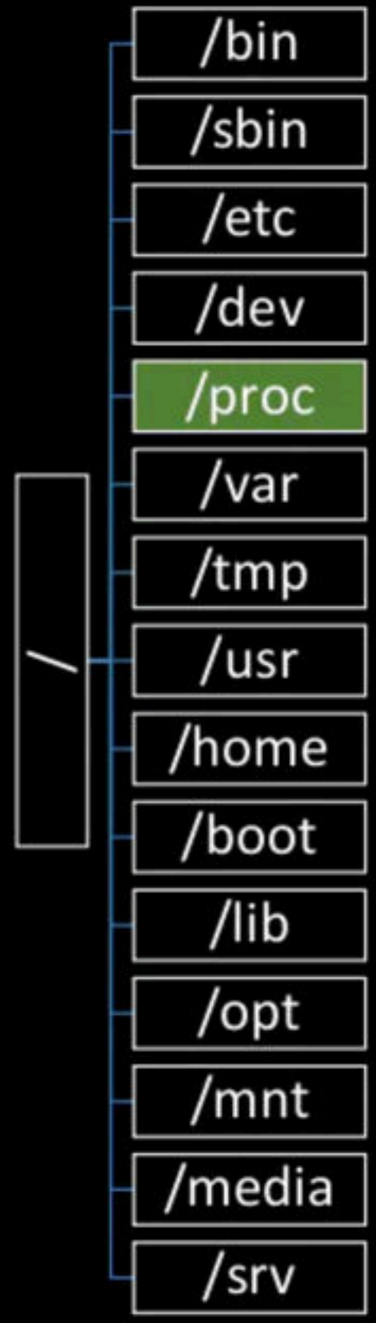
5. /dev – Device Files

- Contains device files.
- These include terminal devices, usb, or any device attached to the system.
- For example: /dev/tty1, /dev/usbmon0



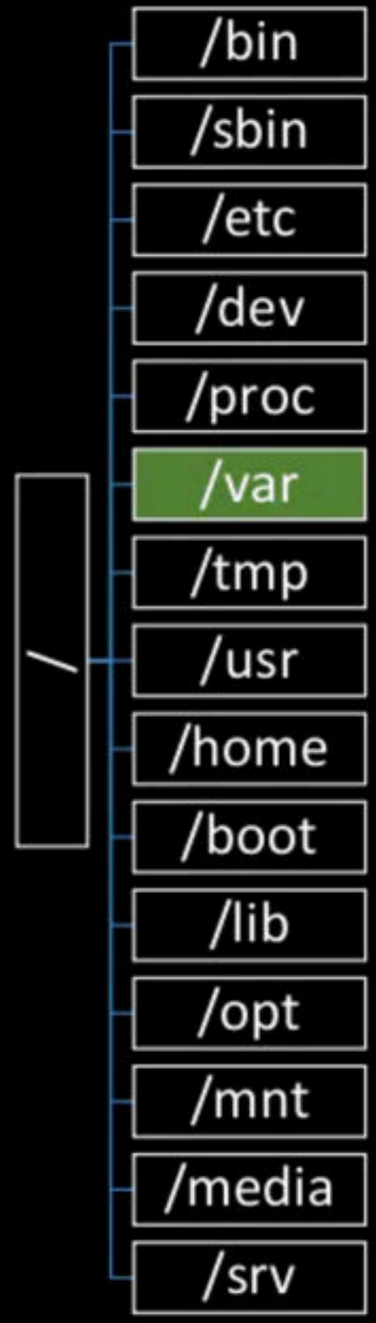
6. /proc – Process Information

- Contains information about system process.
- This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
- This is a virtual filesystem with text information about system resources. For example: /proc/uptime



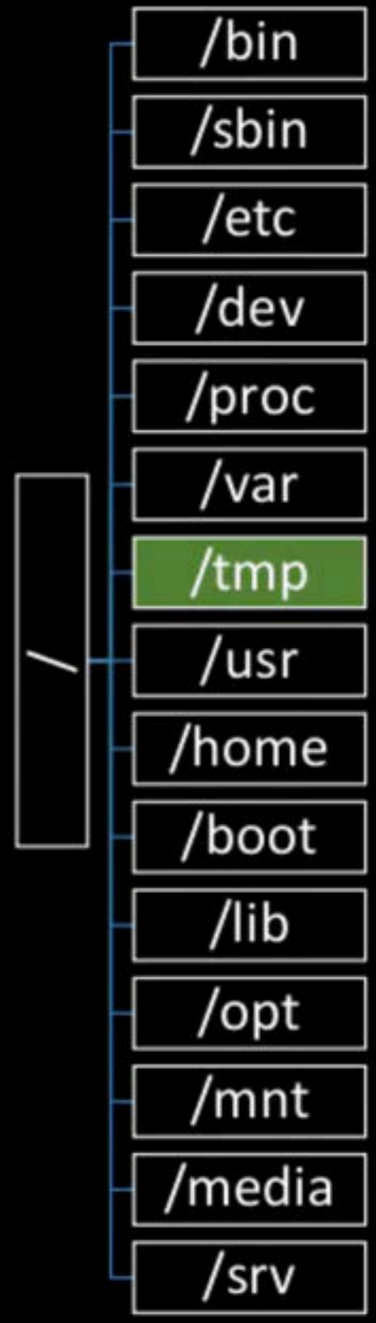
7. /var – Variable Files

- var stands for variable files.
- Content of the files that are expected to grow can be found under this directory.
- This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);



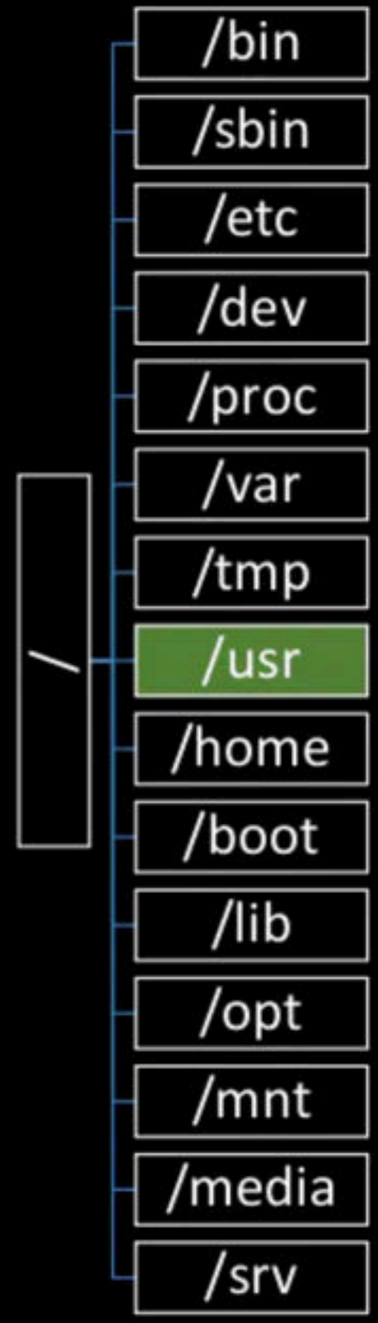
8. /tmp – Temporary Files

- Directory that contains temporary files created by system and users.
- Files under this directory are deleted when the system is rebooted.



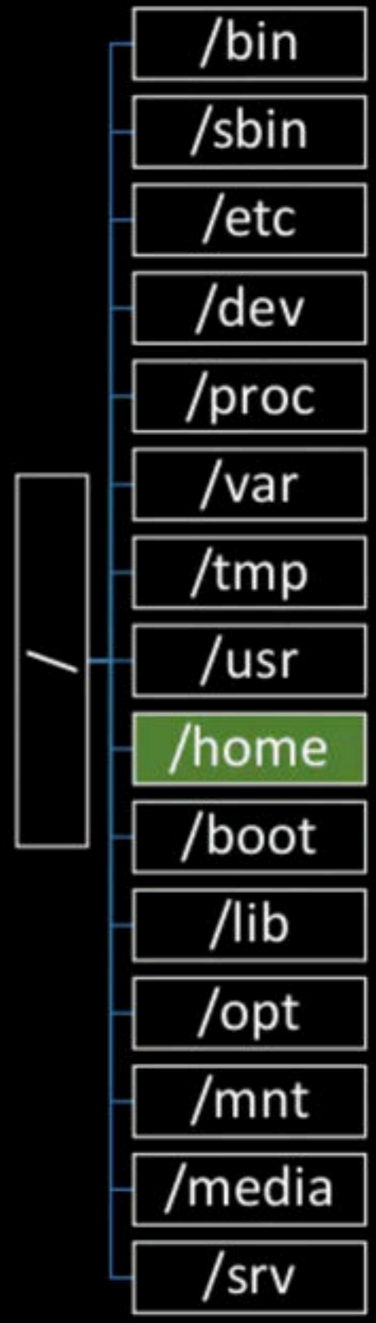
9. /usr – User Programs

- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators.
- If you can't find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2



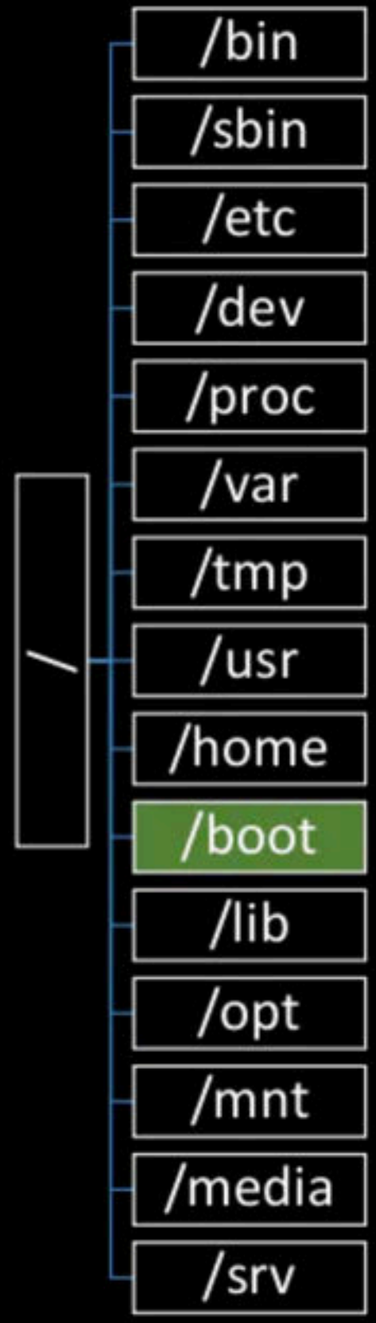
10. /home – Home Directories

- Home directories for all users to store their personal files.
- For example: /home/john, /home/nikita



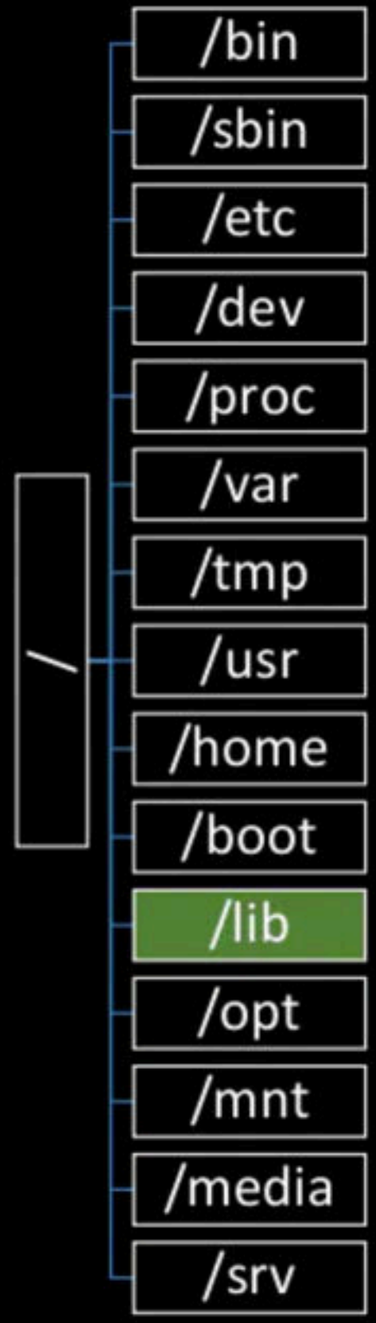
11. /boot – Boot Loader Files

- Contains boot loader related files.
- Kernel initrd, vmlinux, grub files are located under /boot
- For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic



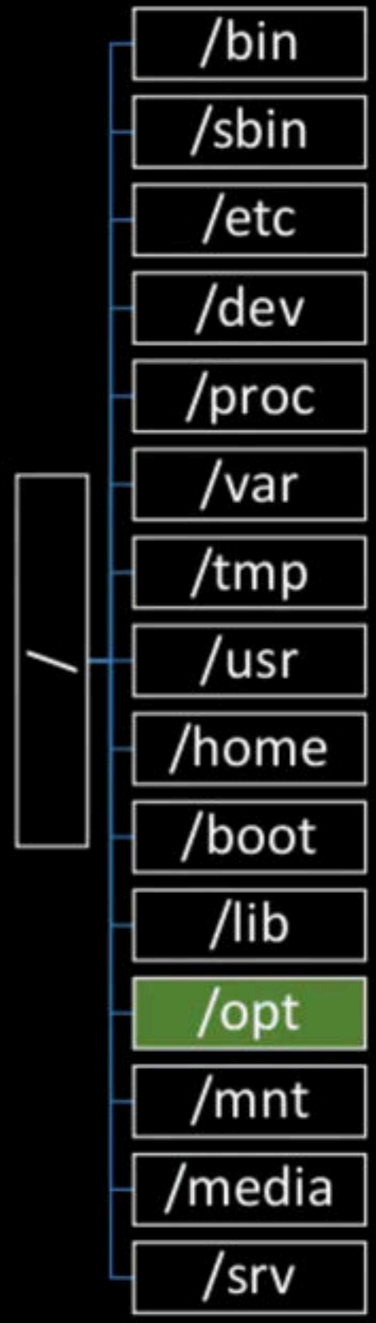
12. /lib – System Libraries

- Contains library files that supports the binaries located under /bin and /sbin
- Library filenames are either ld* or lib*.so.*
- For example: ld-2.11.1.so, libncurses.so.5.7



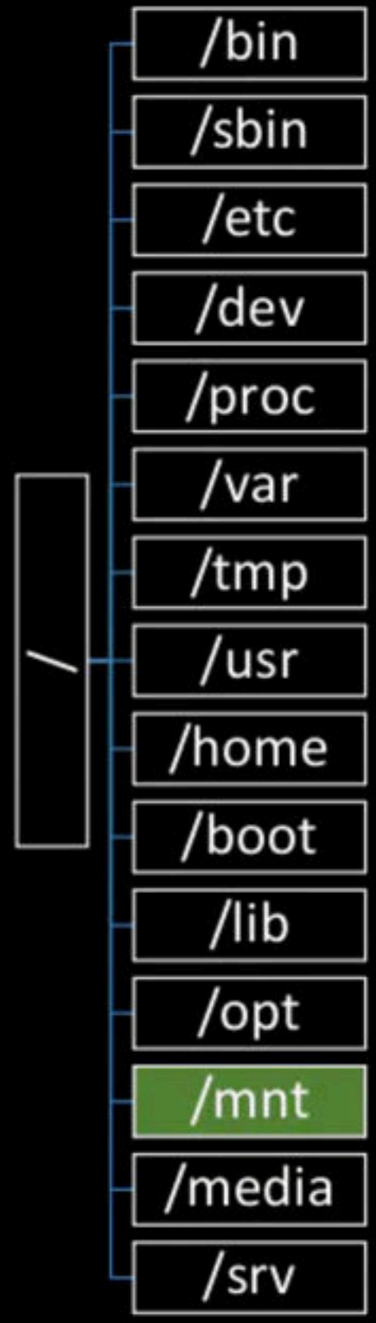
13. /opt – Optional add-on Applications

- opt stands for optional.
- Contains add-on applications from individual vendors.
- add-on applications should be installed under either /opt/ or /opt/ sub-directory.



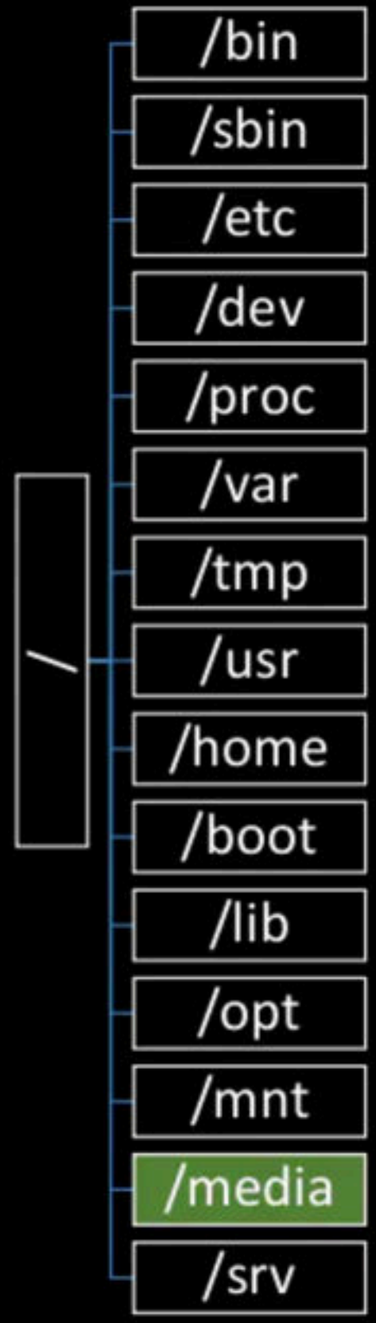
14. /mnt – Mount Directory

- Temporary mount directory where sysadmins can mount filesystems.



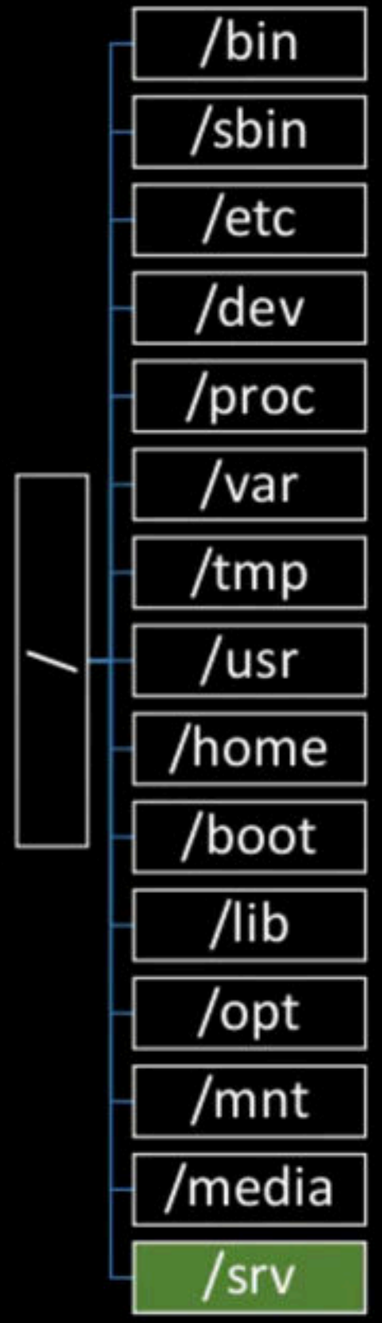
15. /media – Removable Media Devices

- Temporary mount directory for removable devices.
- For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer



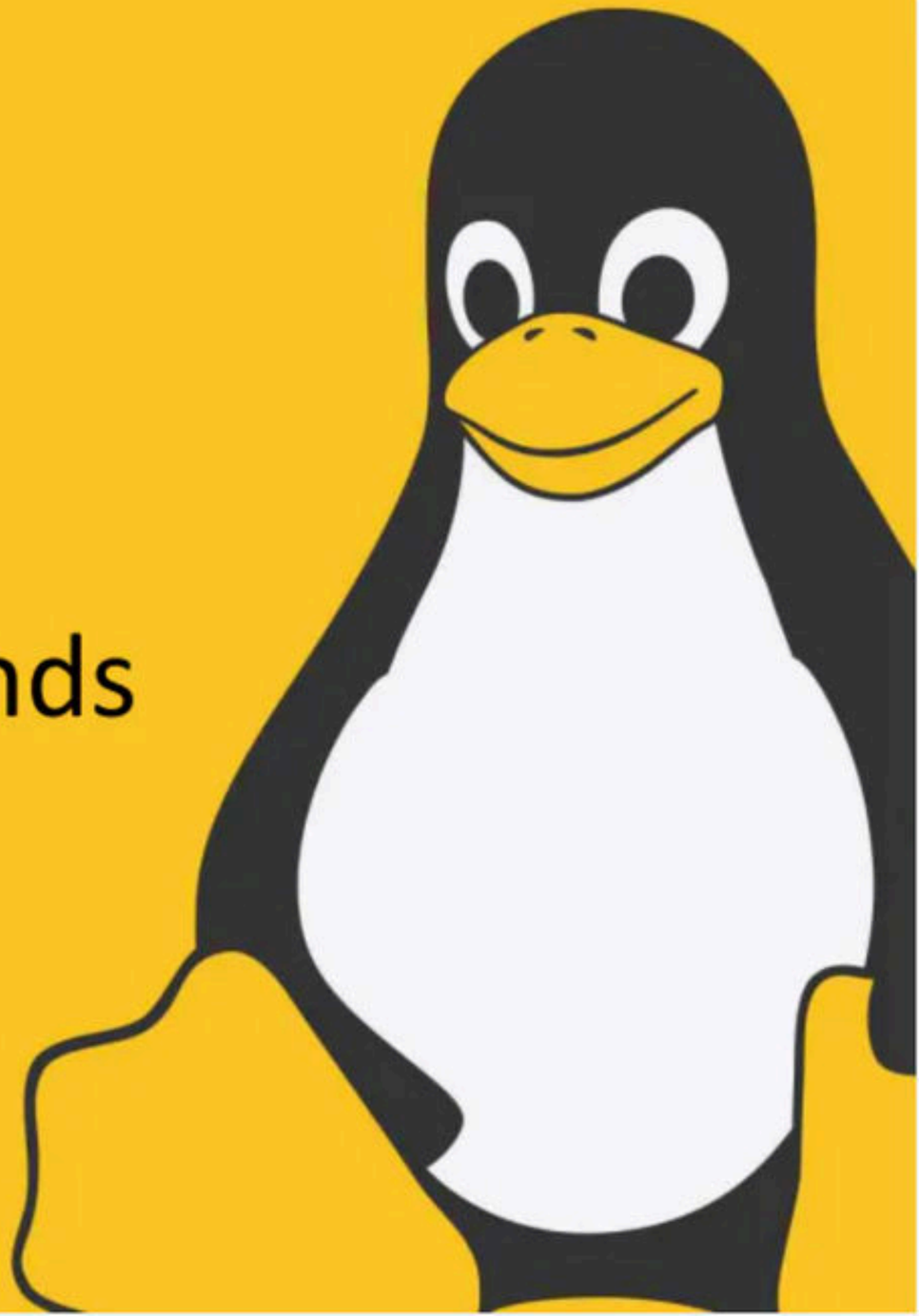
16. /srv – Service Data

- srv stands for service.
- Contains server specific services related data.
- For example, /srv/cvs contains CVS related data.



LINUX
LECTURE 5

File Management Commands
System Calls



Groups

- A group is a collection of users.
- The main purpose of the groups is to define a set of privileges like read, write, or execute permission for a given resource that can be shared among the users within the group.
- A primary group is the default group that a user account belongs to. Every user on Linux belongs to a primary group.
- A user can be part of a maximum 16 groups.

File management related commands:

	Command	Meaning of the command
1	**pwd**	Show the present working directory
2	**ls**	List all the files in the current directory
3	**ls -a**	List the files in the current directory, even the hidden files
4	**ls -t**	List the files sorted by time of modification
5	**cd** DIR	Change directory to folder named DIR
6	**cd**	Change to home directory
7	**mkdir** NEW_FOLDER	Create a directory named NEW_FOLDER
8	**rmdir** NEW_FOLDER	Deletes the directory named NEW_FOLDER
9	**cat FILE1**	Prints the contents of FILE1 . If we give more than one file name, it concatenates them and prints the first file content followed by second file content
10	**head** FILE2	Prints the first 10 lines of FILE2

	Command	Meaning of the command
11	**tail** FILE3	Prints the last 10 lines of FILE3
12	**touch** FILE4	Creates an empty document with name FILE4
13	**rm** FILE5	Deletes the document named FILE5
14	**rm -r** DIR	Deletes the directory DIR
15	**cp** FILE6 FILE7	Copy the contents of FILE6 to FILE7
16	**mv** FILE8 FILE9	Rename or move FILE8 to FILE9
17	**find** FILE10	Search for the FILE10 in the current directory
18	**wc** FILE11	Prints the number of lines, words and byte counts from the FILE11
19	**chown** user FILE12	Change the owner of FILE12 to ‘user’
20	**chgrp** grp FILE13	Change the group ownership of FILE13 to ‘grp’

chmod

- Linux is a multi-user system that uses permissions and ownership for security.
- There are three user types on a Linux system viz. User, Group and Other.
- Linux divides the file permissions into read, write and execute denoted by r,w, and x.
- The command “ls - l” gives the details of permissions that 'user, group, other' have for each of the files in the directory.
- The permissions on a file can be changed by 'chmod' command.

Octal Table for permissions:

binary	octal	permissions
000	0	---
001	1	--x
010	2	-w-
011	3	-wx
100	4	r--
101	5	r-x
110	6	rw-
111	7	rwx

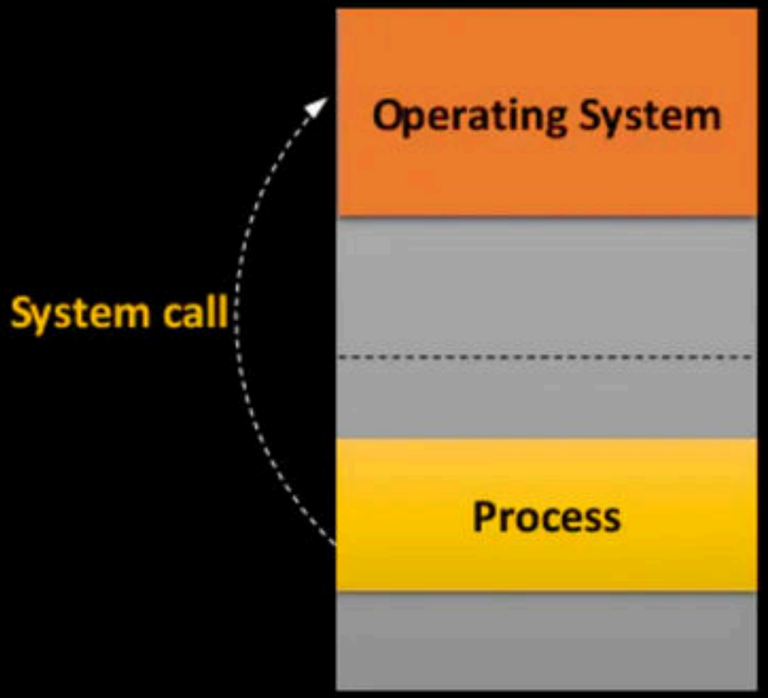
From this we can conclude that, to set r octal will be 4, to set w octal will be 2, to set x octal will be 1. For the User, Group and Other to give all the 3 permissions of read, write and execute we use the code ‘777 which is rwxrwxrwx ‘.

Introduction to System calls:

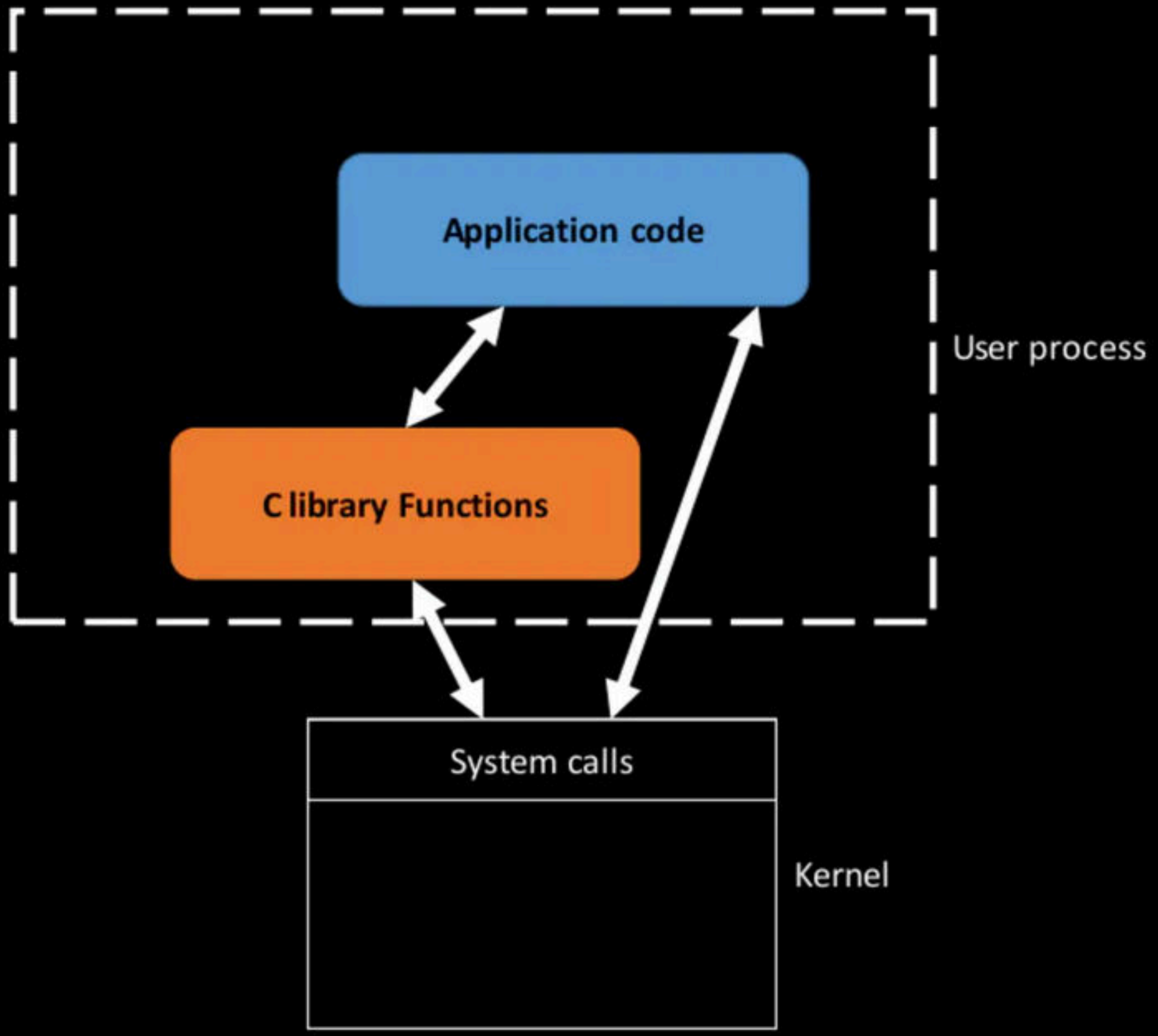
- A system call is just what its name implies -- a request for the operating system to do something on behalf of the user's program.

- The system calls are functions used in the kernel itself.

- To the programmer, the system call appears as a normal C function call. However since a system call executes code in the kernel, there must be a mechanism to change the mode of a process from user mode to kernel mode.



- The C compiler uses a predefined library of functions (the C library) that have the names of the system calls.
- The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls.



LIBRARY FUNCTIONS  SYSTEM CALLS

1.These functions are part of the standard library and can be called from user programs. And will run in user mode. For eg. string manipulation functions like strlen(), strcmp(). System calls are the functions which change the execution mode from user mode to kernel mode. For eg: ‘open()’ is a system call to open a file.

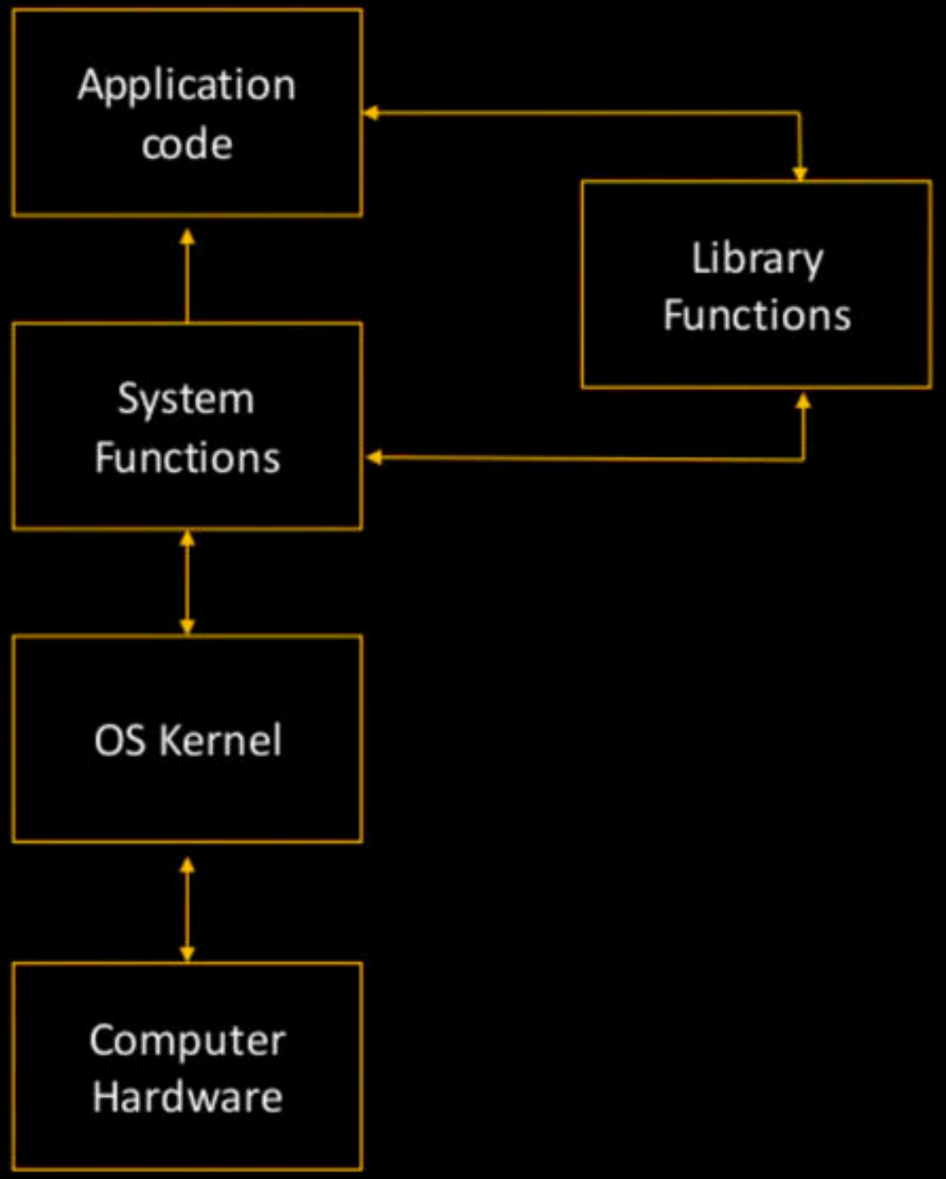
2.There are some library functions which make system calls and some library functions do not make any system calls. String manipulation functions are examples of library functions which don’t make any system calls. Library function ‘fopen()’ which is used for opening a file internally calls the system call ‘open()’.

3.The interaction of different components can be seen below:

To achieve the same task we may have library functions and system calls. A programmer can make use of either the library function calls or system calls. But in such cases it is better to call the library functions because:

i). Library functions will run on all systems but system call may vary from system to system.

ii). By calling the library functions we may reduce the burden of frequent context switches from user mode to kernel mode by using buffering. For eg: fread() to read content from a file will use buffering to prefetch additional data so that the number of system calls to 'read()' are reduced.



Purpose of system calls: Unix system calls are primarily used to **manage the file system** or **control processes** or to **provide communication between multiple processes** or for **device management.**

Purpose	System calls used
Process control	fork(), wait(), exit(), signal(), kill(), exec family of system calls
File manipulation	open(), read(), write(), close(), lseek(), link(), unlink()
Device manipulation	ioctl() along with read(), write()
Time management	time(), gettimer(), settimer(),settimeofday(), alarm()
For Inter process communication	pipe()

System calls for process control:

We will discuss fork() and wait() system calls along with example programs.

fork(), wait() system calls:

Fork system call is used for creating a child process, which runs concurrently with the parent process. After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which are used in the parent process.

Fork doesn't take any parameters but returns an integer value.

Below the return values by fork() with their meaning:

Negative Value: creation of a child process failed.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains the process ID of the newly created child process.

Example program on fork() - DEMO

wait() system call:

Wait system call blocks the parent process until one of its child processes exits or a signal is received. One of the main purposes of wait() is to wait for completion of child processes. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Example program with fork() and wait() – DEMO

Program to demonstrate bottom to up execution of processes using fork() and wait()- DEMO

Exec family of system calls:

Within the process control, exec() family of system calls are very important. We will discuss some of them and run some example programs.

We need to know about the PATH variable which is internally used in running many linux commands and is of specific interest to understand some of the system calls in exec() family.

PATH is an environmental **variable** in **Linux** and other Unix-like operating systems that tells the shell which directories to search for executable files in response to commands issued by a user.

$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
The path names given by PATH environment variable contain the executable files which will be used for running commands.

There are six functions in the exec() family of system calls, there is no exec() system call directly.

1.int execl(const char *path, const char *arg, …, NULL);
2.int execlp(const char *file, const char *arg, …, NULL);
3.int execv(const char *path, char *const argv[]);
4.int execvp(const char *file, char *const argv[]);
5.int execle(const char *path, const char *arg, …, NULL, char * const envp[]);
6.int execvpe(const char *file, char *const argv[], char *const envp[]);

All these functions are used to replace the current process with a new process.

The initial argument for these functions is the name of an executable file that is to be executed.
In the first 3 functions **execl(), execlp(), and execle()** the const char *arg and subsequent list can be considered as a list of arguments arg0, arg1, ..., argn which are passed to the new program to be executed using this system call. Here 'arg0' is the same as the command or program being executed. This list of arguments must end with a NULL pointer as shown in their function prototypes.

In the functions **execv(), execvp(), and execvpe()** also the first parameter is the program being executed and the second parameter gives the list of arguments in an array **argv[].** This array of arguments should also end with a NULL pointer.

In the functions **execlp(), execvp(), and execvpe()** if the first parameter does not contain a path of the executable file, then the executable file is searched in the locations specified in the PATH environment variable.

The **execle() and execvpe()** functions allow the caller to specify the environment of the executed program via the argument **envp**. The envp argument is an array of pointers to strings and must be terminated by a null pointer. The other functions take the environment for the new process image from the external environment variable **environ** in the calling process, but in these two functions it is explicitly passed.

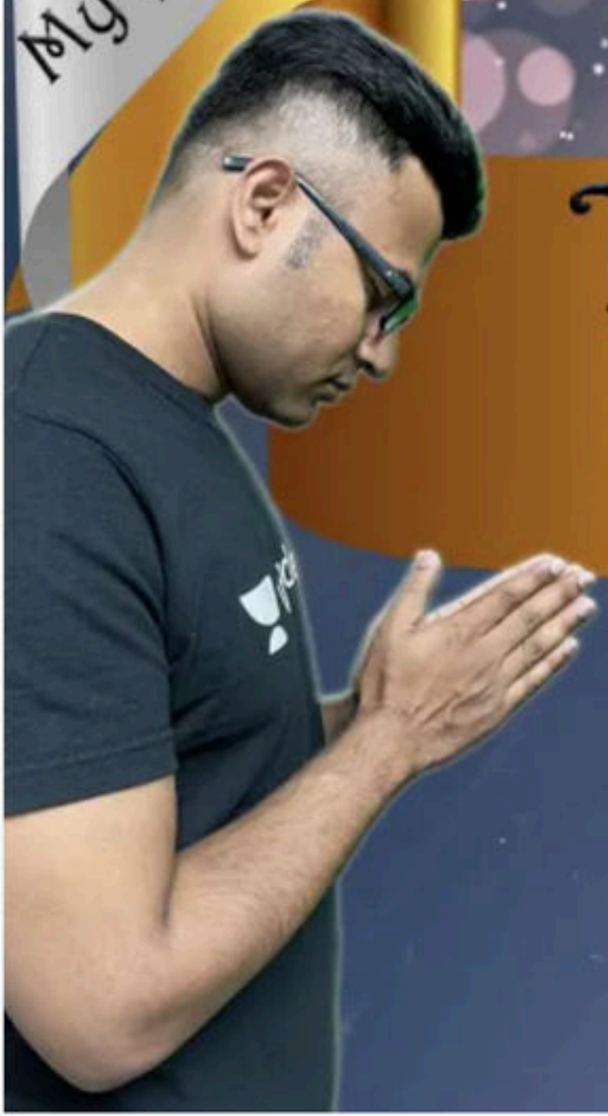
Example Programs: DEMO

1. execl()
2. execlp
3. execv()
4. execvp()

My philosophy

TEACHING IS WORSHIP

STUDENTS ARE GODS



Class will Resume in Few Minutes

Schedule Slide

System calls related to file management:

File descriptor is an integer that uniquely identifies an open file of the process. File descriptor is an important argument in all the file management system calls.

On program startup, the integer file descriptors associated with the streams stdin, stdout, and stderr are 0, 1, and 2 respectively.
The preprocessor symbols STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO are defined with these values in <unistd.h>.

creat System Call:

A file can be created using the **creat** function.
If the **creat()** function runs successfully it returns a file descriptor for the open file, otherwise it returns -1.

At the time of file creation, the following file permissions related to **user**, **group** and **others** can be used. These are defined in the library “sys/stat.h”.

S_IREAD --- read permission for the owner
S_IWRITE --- write permission for the owner
S_IEXEC --- execute/search permission for the owner
S_IRWXU --- read, write, execute permission for the user

S_IRGRP – read for group
S_IWGRP – write for group
S_IXGRP – execute for group
S_IRWXG – read, write, execute for the group

S_IROTH --- read for others
S_IWOTH – write for others
S_IXOTH -- execute for others
S_IRWXO – read , write , execute for others

Example program for creat() - DEMO

Unlink system call:

unlink() accepts a filename as an argument and deletes that file from the file system if that name was the last link to that file and no processes have the file open. The space it was using is made available for reuse after deletion.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.

File open(), read(), write(), close() system calls:

Following are the function prototypes for the file related system calls to open, read, write and close:

int open (const char* Path, int flags);
// First argument is the path to the file which we want to open.
// The second argument ‘flags’ will mention the purpose of opening the file:
O_RDONLY: read only,
O_WRONLY: write only,
O_RDWR: read and write,
O_CREAT: create file if it doesn’t exist,
O_EXCL: prevent creation if it already exists

size_t read (int fd, void* buf, size_t cnt);
// fd is the file descriptor,
// buf is the buffer into which data will be read
//cnt is the number of bytes to be read.
//return value:
return Number of bytes read on success
return 0 on reaching the end of file
return -1 on error
return -1 on signal interrupt

size_t write (int fd, void* buf, size_t cnt);
// fd is the file descriptor
// buf is the buffer which contains the data to be written to the file
// cnt is the number of bytes to be written

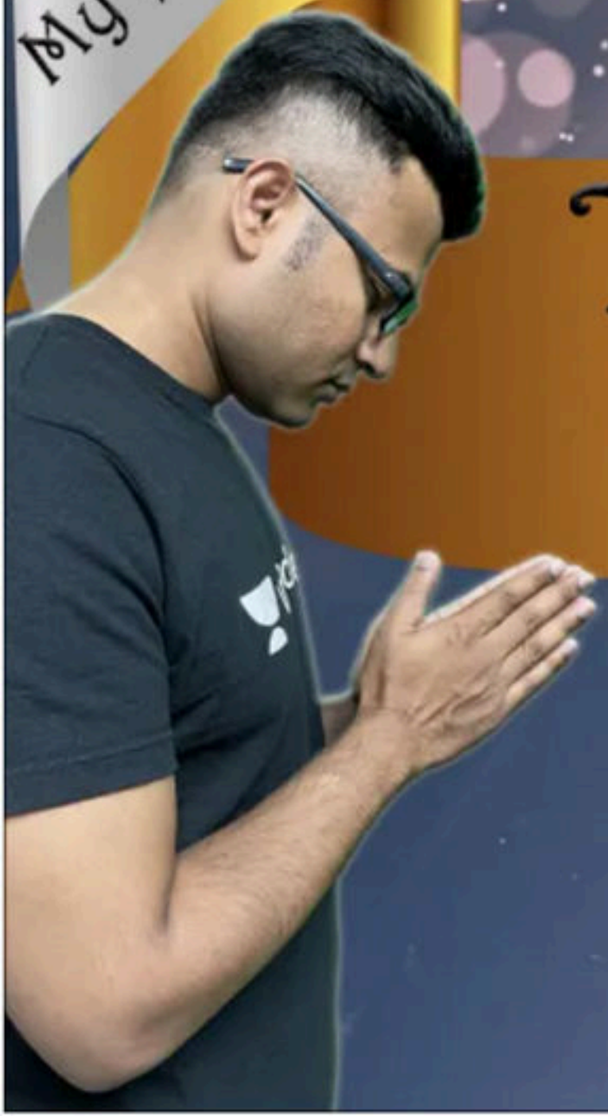
int close(int fd);

Example program for open, write, close system calls - DEMO

My philosophy

TEACHING IS WORSHIP

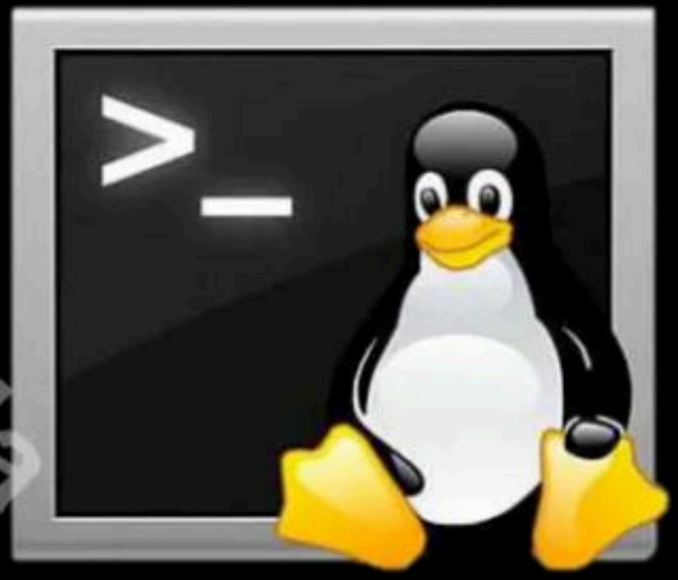
STUDENTS ARE GODS



UNIX SHELL SCRIPTING

Shell:

When a user enters a command in the terminal it goes to the shell which interprets it in a manner the kernel can understand. So, shell is an in-between command interpreter which comes between the user and the kernel.



3 main benefits of a shell

1.When we have to give several file names as arguments to a program or a command, then we can mention them using a pattern or wildcards, the shell will find all the matching filenames.

Ex: If a directory has three files with names file1.txt, file2.txt, file3.txt when we give a command ‘ls *.txt’ this will output all the three files ‘file1.txt, file2.txt, file3.txt’.

2.Input-output redirection.

Using '>' symbol the output of a command can be redirected to a specific file. Similarly using '<' we can give input to a program to be taken from a file.

Ex: In the previous example the output of 'ls *.txt' can be redirected to an output file as below: 'ls *.txt > list_of_files'. Here if the file 'list_of_files' doesn't already exist, then it will be newly created and the output of 'ls *.txt' will be copied into that.

Similarly when reading a huge array of numbers from the user input we can either use file handling functions to read from that input file or we alternatively can use '<' to achieve the same function.

Example program to understand '<' usage for input:

The file 'numbers' contains the list of 10 numbers:
0 31 39 36 35 26 33 39 48 44

This will be given as input to the below program using the '<' symbol.

```c
#include<stdio.h>
int main()
{
  int input[10], i;

  printf("Reading the numbers\n");

  for(i=0; i<10; i++)
  scanf("%d", &input[i]);

  printf("The numbers are\n");
  for(i=0; i<10; i++)
  printf("%d \t", input[i]);
  return 0;
}
```

Execution:

```
$ gcc ip.c
$ ./a.out <numbers
Reading the numbers
The numbers are
0 31 39 36 35 26 33 39 48 44
```

3. Shell allows the users to define their own commands and shortcuts to commands.

Ex: Using the ‘alias’ command we can generate a shortcut to already existing commands.
In the below example we are creating ‘hi’ as a shortcut to ‘ls -lrt’ .

```
$ alias hi='ls -lrt'
```

After this if we type ‘hi’ on the terminal it is the same as typing ‘ls -lrt’ and we will get the same output for both.

What is a command shell?

A program that interprets commands Allows a user to execute commands by typing them manually at a terminal, or automatically in programs called shell scripts.

- A shell is not an operating system.
- It is a way to interface with the operating system and run commands.

Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Linux may use one of the following most popular shells

BASH (Bourne-Again SHell)
CSH (C SHell)
KSH (Korn SHell)

Any of the above shells reads commands from the user (via Keyboard or Mouse) and tells Linux OS what the users want.
If we are giving commands from the keyboard it is called command line interface (Usually in-front of $ prompt, This prompt is dependent upon your shell and Environment that you set or by your System Administrator, therefore you may get a different prompt).

NOTE: To find your shell type following command

echo $SHELL

```
vaanu@vaanu-HP-Notebook:~$ ps
  PID TTY          TIME CMD
 3960 pts/2    00:00:00 bash
 3972 pts/2    00:00:00 ps
vaanu@vaanu-HP-Notebook:~$ echo $SHELL
/bin/bash
vaanu@vaanu-HP-Notebook:~$
```

We can change the shell also

```
vaanu@vaanu-HP-Notebook:~/Videos$ echo $SHELL
/bin/bash
vaanu@vaanu-HP-Notebook:~/Videos$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
vaanu@vaanu-HP-Notebook:~/Videos$ chsh
Password:
Changing the login shell for vaanu
Enter the new value, or press ENTER for the default
        Login Shell [/bin/sh]: /bin/sh
vaanu@vaanu-HP-Notebook:~/Videos$ echo $SHELL
/bin/bash
vaanu@vaanu-HP-Notebook:~/Videos$ su - vaanu
Password:
$ echo $SHELL
/bin/sh
$
```

Echo $SHELL
Shows my current shell is /bin/bash

$cat /etc/shells
Shows the available shell in system

$chsh
change shell
Entered new shell /bin/sh
Now we need to su login (super user login to reflect the changes in shell)
$su - vaanu (my account name)
Now we can see our new running shell /bin/sh

```
vaanu@vaanu-HP-Notebook:~/Videos$ echo $SHELL
/bin/bash
vaanu@vaanu-HP-Notebook:~/Videos$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
vaanu@vaanu-HP-Notebook:~/Videos$ chsh
Password:
Changing the login shell for vaanu
Enter the new value, or press ENTER for the default
        Login Shell [/bin/sh]: /bin/sh
vaanu@vaanu-HP-Notebook:~/Videos$ echo $SHELL
/bin/bash
vaanu@vaanu-HP-Notebook:~/Videos$ su - vaanu
Password:
$ echo $SHELL
/bin/sh
$
```

What is BASH?

BASH = Bourne Again SHell Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems. It has all of the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line. Since it is Free Software, it has been adopted as the default shell on most Linux.

If we type **ps command**

ps command is used to list the currently running processes and their PIDs along with some other information depends on different options.

```
vaanu@vaanu-HP-Notebook:~$ ps
  PID TTY          TIME CMD
 3960 pts/2    00:00:00 bash
 3972 pts/2    00:00:00 ps
vaanu@vaanu-HP-Notebook:~$
```

It will show bash shell is running.
The output includes information about the shell (bash) and the process running in this shell (ps, the command that you typed):
When we login in system and until we logout, the shell (bash shell here) is running all the time.

Shell scripting

A shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

When a group of commands have to be executed regularly, they should be stored in a file, and the file itself executed as shell script or shell program.

Though it is not mandatory, we normally use the **.sh** extension for shell scripts.

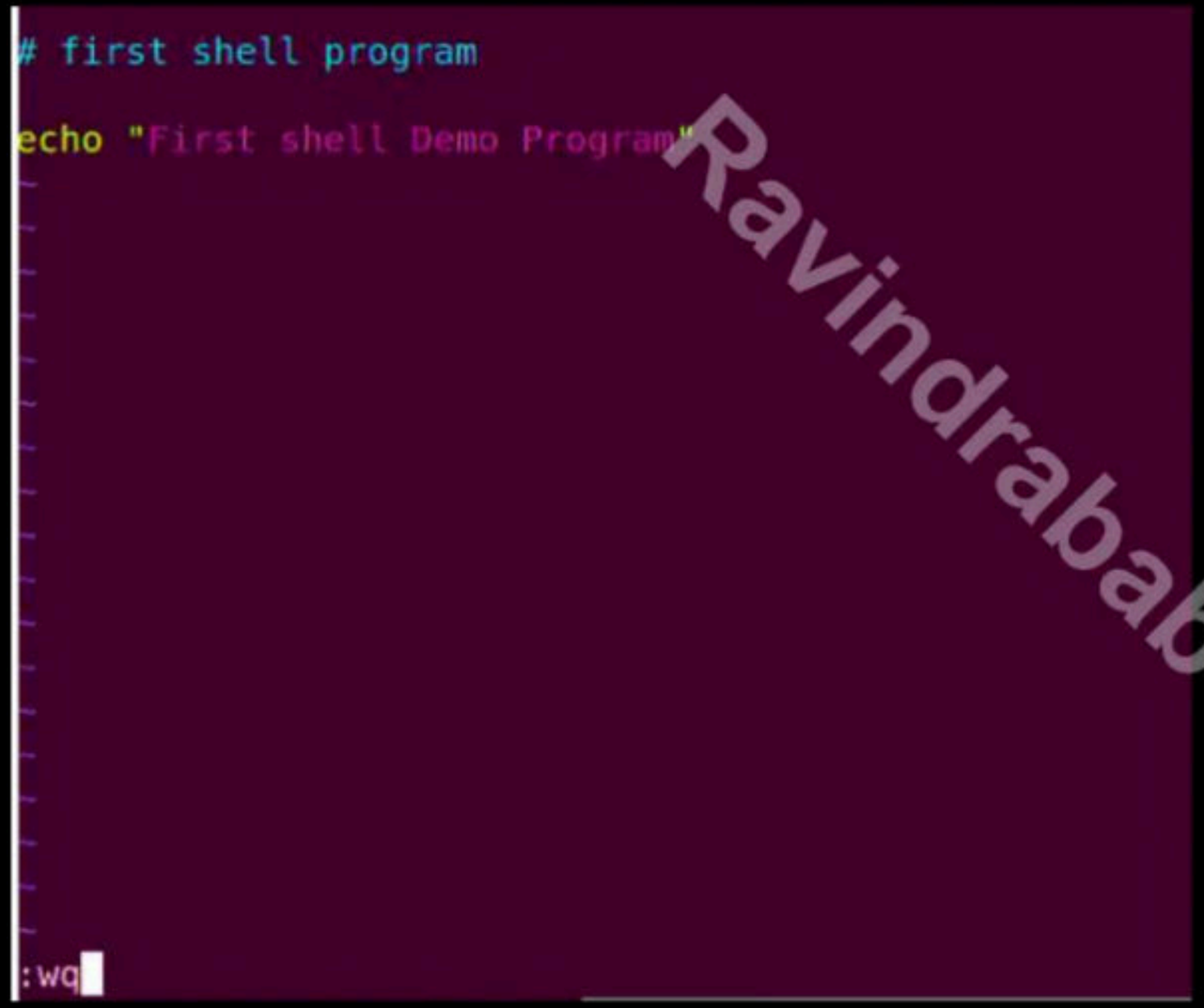
A shell script is a file that contains ASCII text.
To create a shell script, we use a *text editor*.

A text editor is a program, like a word processor, that reads and writes ASCII text files. There are many, many text editors available for Linux systems, both for the command line and GUI environments. Here is a list of some common ones:

vi
vim /* vim may require to install */
gedit

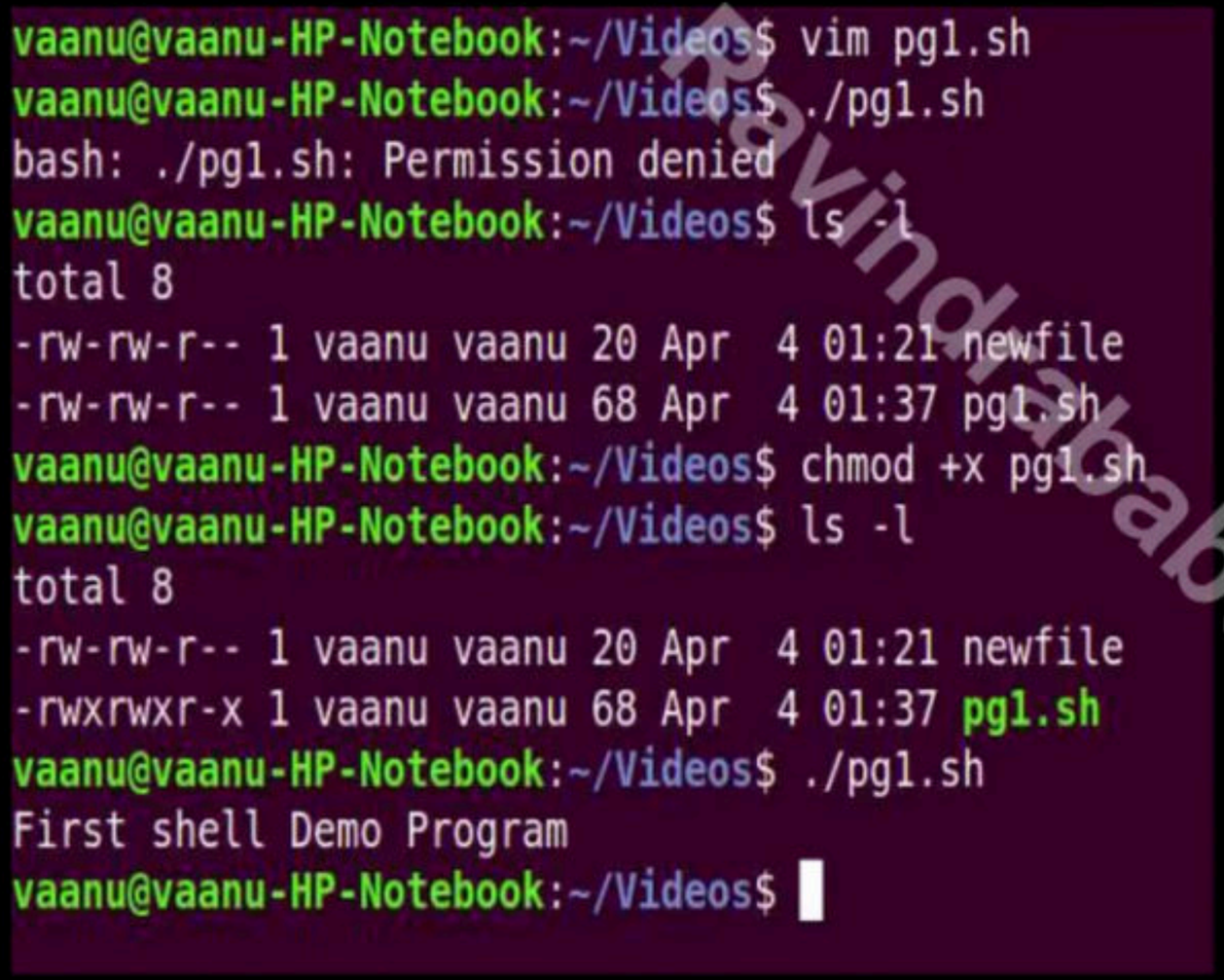
Command to install vim
$sudo apt install vim

Run command
$vim pg1.sh



Before writing program press “i” to come in insert mode
And to save
:wq

Any line starting with # is considered as a comment in shell script.



When we try to run program first time by command ./pg1.sh

It says permission denied as no execute permission to pg1.sh

So add execute permission by chmod +x pg1.sh

Then again run the program ./pg1.sh

Regular expressions:

A set of special characters used to check patterns in strings are called regular expressions.
In short they are also called 'regexp' and 'regex'.
They are useful in writing scripts.

Listed below are some of the commonly regular expression symbols with their usage:

Symbol	Description
^	Matches starting of the string
$	Matches end of the string
?	Matches exactly one character
*	Matches zero more times occurrence of the preceding character
\	To represent special characters
()	To group regular expressions
\+	Matches one or more of the previous character
\?	Matches zero or one occurrence of the previous character

We will see examples of how regular expressions are useful together with the ‘grep’ command. ‘grep’ is used for printing lines from an input file which are matching a given pattern.

Let us say we have a file with the name ‘cities’ having the list of top 10 highly populated cities in India:

$ cat cities

Mumbai
Delhi
Bangalore
Hyderabad
Ahmedabad
Chennai
Kolkata
Surat
Pune
Jaipur

We can search for the cities having letter ‘b’ anywhere in the name. For this we will use **pipe character ‘|’ , it redirects the output of one command into another command**

$ cat cities | grep b
Mumbai
Hyderabad
Ahmedabad

In this above example we have directed the output of ‘cat cities’ to the ‘grep’ command and found the cities which have ‘b’ in their names.

To print the list of cities starting with ‘A, or B, C’:

$cat cities | grep ^[A,B,C]

Bangalore

Ahmedabad

Chennai

To print the list of cities ending with ‘i’:

$ cat cities | grep i$
Mumbai
Delhi
Chennai

Search for a city name having one or more **n**'s

$ cat cities_1 | grep 'n\+'

Bangalore
Chennai
Pune

Similarly all the other symbols can be tried.

Control statements and loops:

1.In shell we have **if** and **case** control statements which check for a condition and run based on the yes/no output. **'case'** statement works exactly like the **switch** statement of C.
2.To run the same set instructions for several iterations we have **for** loops**, while..do** loops and **Until..do** loops.
3.There are **break** and **continue** commands which work exactly like in C.

Syntax of "if" :

Syntax 1

```
if [condition]
then
    execute  commands
else
     Execute commands
fi
```

Syntax 2

```
if [condition]
then
    execute  commands
fi
```

Syntax 3

```
if [condition]
then
    execute  commands
elif [condition]
then
     Execute commands
else
Execute commands
fi
```

Syntax of case:

```
case  in
  Pattern 1) Statement 1;;
  Pattern 2) Statement 2;;
  Pattern n) Statement n;;
esac
```

Syntax 1 of for loop:

```
for arg in list
 do
        command(s)
  ...
 done
```

Example:

```
for i in 1 2 3 4
do
echo " $i times 2 is $(( $i * 2 )) "
done
```

Syntax 2 of for loop:

There is a C like syntax of for loop in shell also.

```
for (( initialize ; test ; increment ))
do
    commands
done
```

Syntax of “while”

```
while condition is true
do
    commands
done
```

Shell relative operators:

- **The syntax to be followed:** The conditional expressions should be placed inside square brackets, and there should be space between the square bracket and the variable.

- **[$a < $b]** is wrong as there is no space between the square brackets and the operands.

- **[$a < $b]** is correct as there is space between the square bracket **[** and the operand $a.. Similarly there is a space between $b and **].**

- We can use the mathematical symbols of <, >, = or we can also use operators like , **-lt, -gt** and **-eq** to write the conditional expressions.
- We can look at different relative operators with examples.

Operator	Description	Example for x=5, y=10
-eq	The condition is true if the values of the two operands are equal otherwise it is false.	[$x -eq $y] is false as x is not equal to y.
-ne	The condition is true if the values of the two operands are not equal otherwise it is false.	[$x -ne $y] is true as x is not equal to y.
-gt	The condition is true if the value of the left operand is greater than that of right operand otherwise it is false.	[$x -gt $y] is false as x is not greater than y.
-lt	The condition is true if the value of the left operand is less than that of right operand otherwise it is false.	[$x -lt $y] is true as x is less than y.
-ge	The condition is true if the value of the left operand is greater than or equal to that of right operand otherwise it is false.	[$x -ge $y] is not true as x is less than y.
-le	The condition is true if the value of the left operand is less than or equal to that of right operand otherwise it is false.	[$x -le $y] is true as x is less than y.

‘wc’ command to count number of files in current directory:

Program name pg2.sh

#!/bin/bash

val=`ls|wc -l`

echo There are total $val files in current directory


```
#!/bin/bash

val=`ls|wc -l`

echo There are total $val files in current directory
```

‘wc’ command to count number of files in current directory:

Program name pg2.sh

```
#!/bin/bash

val=`ls|wc -l`

echo There are total $val files in current directory
```


Explanation of constituents in program
Back quote -> symbol
Below (esc key in keyboard)

The back quote assigns output from system commands to variables
For ex: The output of `ls | wc -l` is assigned to variable val

```
#!/bin/bash

val=`ls|wc -l`

echo There are total $val files in current directory
```


val is a variable which is assigned with output produced by backquote

A variable's value is obtained by the prefixing it with the $ character
For ex: echo $val will print 5
wc stands for wordcount. For ex:
If I give command
wc pg2.sh

It will display four things
Number of lines (including blank line so 5), number of words in file (12) (space is considered as delimiter) so
val=`ls|wc is one word and -l` is second word (in 3rd line)
Total 82 characters

Please note
#!/bin/bash is actually 11 characters but at the end a newline character is assumed so total 12 charatecers is assumed by wc command, same for each line, if line is blank (for ex 2nd line) then also one character is assumed. Space is also one character.
Last column (4th word in output) represent the file name given as argument

```
vaanu@vaanu-HP-Notebook:~/Videos$ cat pg2.sh
#!/bin/bash

val=`ls|wc -l`

echo There are total $val files in current directory
vaanu@vaanu-HP-Notebook:~/Videos$ wc pg2.sh
 5 12 82 pg2.sh
```


wc -l gives only number of lines in a file
wc -w gives only number of words in a file
wc -c gives only number of characters in a file

Coming to program

```
vaanu@vaanu-HP-Notebook:~/Videos$ vim pg2.sh
vaanu@vaanu-HP-Notebook:~/Videos$ chmod +x pg2.sh
vaanu@vaanu-HP-Notebook:~/Videos$ ./pg2.sh
There are total 5 files in current directory
vaanu@vaanu-HP-Notebook:~/Videos$ ls
newfile  pg1.sh  pg2.sh  videoplayback1.mp4  videoplayback.mp4
vaanu@vaanu-HP-Notebook:~/Videos$
```


Please note: After writing program we need to assign execute permission to program
Hence chmod +x pg2.sh

Step1: The output of ls will redirect to wc -l
Output of ls will be (as can be seen in above screenshot) all files in directory
wc -l will count the number of lines (one file name is one line)

As using ls we can see there are 5 files in current directory so the program prints
There are a total of 5 files in the current directory.

In aforementioned program st
Val is a variable.
More details about variables in shell script:
Shell variables do not need to be pre declared. Shell variables are created by simply assigning with them .
The shell considers all variables to contain strings of characters and so no type information is required.

Note: if we have to give two values separated by space then we have to give in double quotes so that the shell treats it as one variable.

For ex:
string1=HELLO
It will work
But
string1=HELLO WORLD
It will not work
We need to give in double quotes

```
vaanu@vaanu-HP-Notebook:~/Videos$ string1=HELLO
vaanu@vaanu-HP-Notebook:~/Videos$ echo $string1
HELLO
vaanu@vaanu-HP-Notebook:~/Videos$ string1=HELLO WORLD
WORLD: command not found
vaanu@vaanu-HP-Notebook:~/Videos$ string1="HELLO WORLD"
vaanu@vaanu-HP-Notebook:~/Videos$ echo $string1
HELLO WORLD
```

Also there should be no space for ex:

string1=HELLO will work but

string1 = HELLO

Will not work (as space is there)

In first case shell will see = and treat the command as variable assignment while in second case string1 will be assumed as some command

```
vaanu@vaanu-HP-Notebook:~/Videos$ string1="HELLO WORLD"
vaanu@vaanu-HP-Notebook:~/Videos$ echo $string1
HELLO WORLD
vaanu@vaanu-HP-Notebook:~/Videos$ string1 ="HELLO WORLD"
No command 'string1' found, did you mean:
 Command 'strings' from package 'binutils-multiarch' (main)
 Command 'strings' from package 'binutils' (main)
string1: command not found
```

In below program,
readonly <variable_name >
Will make the variable as constant and cannot reassign to some other value again


```
vaanu@vaanu-HP-Notebook:~/Videos$ value=10
vaanu@vaanu-HP-Notebook:~/Videos$ echo $value
10
vaanu@vaanu-HP-Notebook:~/Videos$ value=11
vaanu@vaanu-HP-Notebook:~/Videos$ echo $value
11
vaanu@vaanu-HP-Notebook:~/Videos$ readonly value
vaanu@vaanu-HP-Notebook:~/Videos$ value=12
bash: value: readonly variable
vaanu@vaanu-HP-Notebook:~/Videos$ string="Hello Linux"
vaanu@vaanu-HP-Notebook:~/Videos$ echo $string
Hello Linux
vaanu@vaanu-HP-Notebook:~/Videos$ string="Hello Linux shell script"
vaanu@vaanu-HP-Notebook:~/Videos$ echo $string
Hello Linux shell script
vaanu@vaanu-HP-Notebook:~/Videos$ readonly string
vaanu@vaanu-HP-Notebook:~/Videos$ string="Hello Linux shell script program"
bash: string: readonly variable
```

expr command:

expr command evaluates the current expression and displays the corresponding output
Note: Both values should be integer

```
vaanu@vaanu-HP-Notebook:~$ expr 20 + 21
41
vaanu@vaanu-HP-Notebook:~$ expr 20 - 10
10
vaanu@vaanu-HP-Notebook:~$ expr 20 - 21
-1
vaanu@vaanu-HP-Notebook:~$ expr 20 \* 2
40
vaanu@vaanu-HP-Notebook:~$ expr 20 / 5
4
```

Note: expr 20 / 3 will give 6   [decimal portion is truncated]

expr commands combine two functions
1.Perform arithmetic operation on integers
2.Manipulate strings

For manipulating strings, expr uses two expressions separated by a colon.
The string to be worked upon is placed on the left of the : and a regular expression is placed on its right.

Depending on the composition of the expression, expr can perform three important string functions

1. Determine the length of string
2. Extract a substring
3. Locate the position of a character in a string

1.Determining the length of string:

$ expr "abcdefghijklmnopqrstuvwxyz" : '.*'

Here .* is in single quote

The regular expression .* signifies to expr that it has to print the number of characters matching the pattern, i.e, length of the entire string

. (dot) means any one character except null character

* Means any number of times (0 or more)

Combinely .* means any character any number of times till new line (\n) is encountered.

```
vaanu@vaanu-HP-Notebook:~$ expr "abcdefghijklmnopqrstuvwxyz" : '.*'
26
```

2. Extracting a substring

$ expr "ABCDEF" : '...\(...\)'
Op: DEF

...\(...\) it signifies that first three characters (three dots) to be ignored and remaining three characters to be extracted from 4th position

```
vaanu@vaanu-HP-Notebook:~/Videos$ expr "ABCDEF" : '...\(...\)'
DEF
vaanu@vaanu-HP-Notebook:~/Videos$ expr "ABCDEF" : '...\(..\)'
DE
vaanu@vaanu-HP-Notebook:~/Videos$ expr "ABCDEF" : '..\(..\)'
CD
```

Demo program for ‘while’, ‘if’ and ‘-gt’:

To check the entered name of a person is less than 5 characters in length

```
while echo "enter name:"
do
        read name
       value=`expr $name : '.*'`

        reqlen=5

       if [ $value -gt $reqlen ]
       then
            echo "Name $name has length
       $value  greater than 5"
        else
           break
        fi

done
```

Demo program for ‘while’, ‘if’ and ‘-gt’:

To check the entered name of a person is less than 5 characters in length

```
while echo "enter name:"
do
        read name
       value=`expr $name : '.*'`

        reqlen=5

       if [ $value -gt $reqlen ]
       then
            echo "Name $name has length
       $value  greater than 5"
        else
           break
        fi

done
```


Analysis:

In while the condition is echo which always return true
The entered value is read in variable name [read is like scanf]

`` `expr $name : '.*'` `` This will check the number of characters in name (as explained in expr command)
The output of above will be saved in variable value

reqlen =5 /* reqlen is a variable initialized with 5 , reqlen: required length]

if [$value -gt $reqlen]
Note [ ] is used to evaluate the expression value > reqlen and it is used with if and other place also (like while), In our while we have single argument echo so did not used [ ]
-gt is an operator greater than
So if the value > reqlen
Print name has length greater than 5
If value is less than reqlen then else part will execute and break statement will break the while loop

```
while echo "enter name:"
do
 read name
value=`expr $name : '.*'`

 reqlen=5

if [ $value -gt $reqlen ]
then
      echo "Name $name has length $value  greater than 5"
 else
    break
 fi

done
```

```
vaanu@vaanu-HP-Notebook:~/Videos$ vim pg3.sh
vaanu@vaanu-HP-Notebook:~/Videos$ chmod +x pg3.sh
vaanu@vaanu-HP-Notebook:~/Videos$ ./pg3.sh
enter name:
sachin
Name sachin has length 6  greater than 5
enter name:
sehwag
Name sehwag has length 6  greater than 5
enter name:
virat
vaanu@vaanu-HP-Notebook:~/Videos$
```

tee command

- tee is a command which handles character stream by duplicating its input. It saves one copy in a file and writes the other to standard output.
- **tee is mostly used in combination with other commands through piping.**

Demo on tee command

```
vaanu@vaanu-HP-Notebook:~/Videos$ ls
newfile  pg1.sh  pg2.sh  pg3.sh  videoplayback1.mp4  videoplayback.mp4
vaanu@vaanu-HP-Notebook:~/Videos$ ls | tee myfile.txt
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
```


ls command shows the files in my current directory

ls | tee myfile.txt

In above command, tee is used with ls through piping. The output of ls is redirected to tee and tee displays (writes) on standard output (terminal) and also write it on myfile.txtEarlier we can see there is no file named with myfile.txt in directory, but now it is added and we can see the content of file using cat commad. Please note first ls doesnot show myfile.txt but after running tee command myfile.txt is present in terminal, so we can see that tee first writes the output in myfile.txt and then displays on standard output (terminal)

Again using ls and cat myfile.txt the output is

```
vaanu@vaanu-HP-Notebook:~/Videos$ ls
myfile.txt   pg1.sh  pg3.sh                 videoplayback.mp4
newfile      pg2.sh  videoplayback1.mp4
vaanu@vaanu-HP-Notebook:~/Videos$ cat myfile.txt
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
```

Important question (can be asked in interview)

How can we use **tee command** to display both the list of files and its count on terminal?

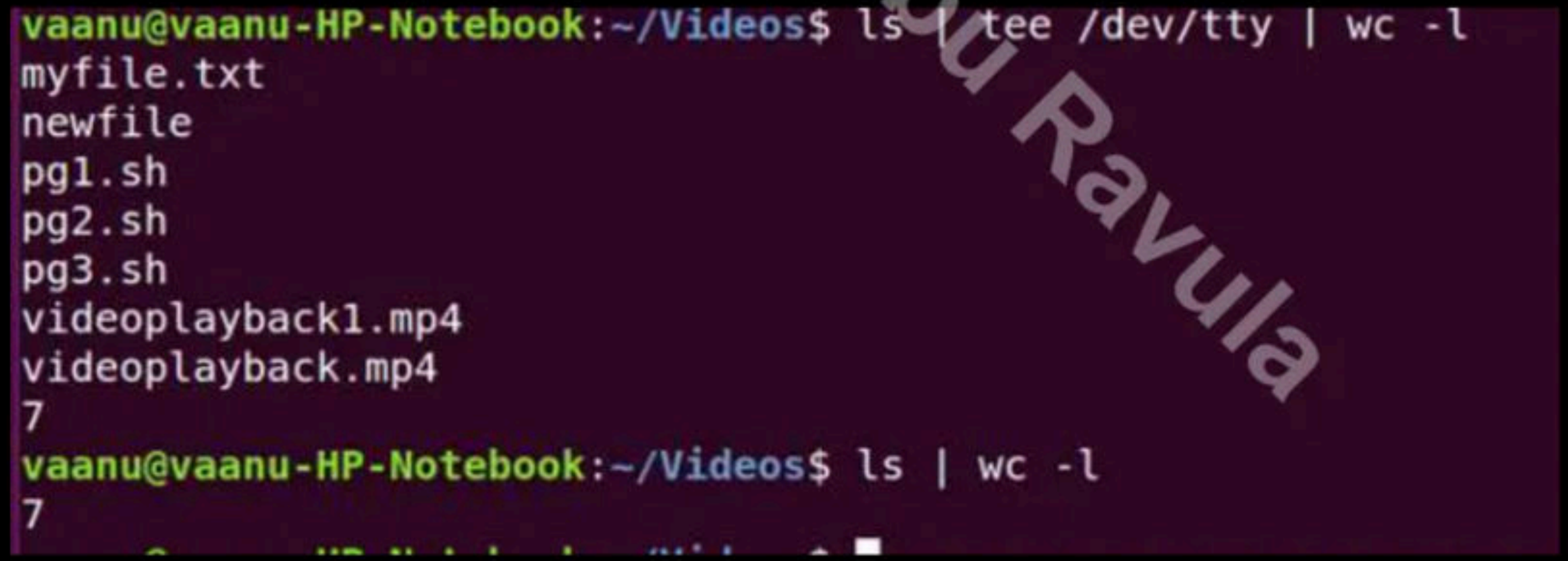
$ ls | tee /dev/tty | wc -l

```
vaanu@vaanu-HP-Notebook:~/Videos$ ls | tee /dev/tty | wc -l
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
7
```

Please note : in previous command we used

tee /dev/tty (in place of using some file namd such as myfile.txt as in previous case, we used /dev/tty which is nothing but the terminal as terminal is also a file in linux. /dev/tty is a special file which indicates someone terminal, for example we can use command $**who >/dev/tty** and it will display the users terminal currently he is using along with date and time of login) so here output of ls is redirected to tee command and its output is again redirected to wc command (so at bottom we can see 7)

If we do not use tee command then it will only display the number of files (for ex below)



Example 2:

We can use df-h command to display the disk usage and writes it on some file using tee command

```
vaanu@vaanu-HP-Notebook:~/Videos$ df -h | tee diskusagefile.txt
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G     0  3.9G   0% /dev
tmpfs           788M  9.3M  779M   2% /run
/dev/sda5       241G   41G  188G  18% /
tmpfs           3.9G   44M  3.9G   2% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
tmpfs           788M   76K  788M   1% /run/user/1000
```

tee command syntax

tee [options] [file]

-a , this is used for appending the content in an already existing file, if we do not use -a option then tee will overwrite in the file


```
vaanu@vaanu-HP-Notebook:~/Videos$ cat diskusagefile.txt
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G     0  3.9G   0% /dev
tmpfs           788M  9.3M  779M   2% /run
/dev/sda5       241G   41G  188G  18% /
tmpfs           3.9G   44M  3.9G   2% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
tmpfs           788M   76K  788M   1% /run/user/1000
vaanu@vaanu-HP-Notebook:~/Videos$ ls | tee -a diskusagefile.txt
diskusagefile.txt
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
```

Now again displaying the content of disk usage

```
vaanu@vaanu-HP-Notebook:~/Videos$ cat diskusagefile.txt
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G     0  3.9G   0% /dev
tmpfs           788M  9.3M  779M   2% /run
/dev/sda5       241G   41G  188G  18% /
tmpfs           3.9G   44M  3.9G   2% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
tmpfs           788M   76K  788M   1% /run/user/1000
diskusagefile.txt
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
```


We can see the output of df -h and output of ls both are in file.

tee command can also write in multiple files

$ ls | tee myfile1.txt myfile2.txt myfile3.txt


```
vaanu@vaanu-HP-Notebook:~/Videos$ ls
diskusagefile.txt  myfile.txt  newfile  pg1.sh  pg2.sh  pg3.sh  videoplayback1.mp4  videoplayback.mp4
vaanu@vaanu-HP-Notebook:~/Videos$ ls | tee file1.txt file2.txt file3.txt
diskusagefile.txt
myfile.txt
newfile
pg1.sh
pg2.sh
pg3.sh
videoplayback1.mp4
videoplayback.mp4
vaanu@vaanu-HP-Notebook:~/Videos$ ls
diskusagefile.txt  file1.txt  file2.txt  file3.txt  myfile.txt  newfile  pg1.sh  pg2.sh  pg3.sh  videoplayback1.mp4  videoplayback.mp4
```

Linux Processes:

A process is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions (called "text"), data, and stack.

On a UNIX system every process except the process 0 is created when another process executes the fork system call. The kernel identifies each process by its process ID (PID).

Process 0 is a special process that is created when the system boots; after forking a child process (process 1), process 0 becomes the **swapper process** which is responsible for swapping processes in and out of main memory.

Process 1, known as **init,** is the ancestor of every other process in the system.

The **executable file** created after compilation consists of several parts like:

1. A set of "headers" that describe the attributes of the file
2. The program text or code
3. A machine language representation of data that has initial values when the program starts execution, and an indication of how much space the kernel should allocate for uninitialized data, called bss (the kernel initializes it to 0 at run time)
4. other sections, such as symbol table information.

In UNIX there are **kernel stack** and **user stack** for the processes running in the kernel mode and user mode respectively.

The user stack contains the arguments, local variables, and other data for functions executing in user mode.

The kernel stack also contains similar information for the functions running in kernel mode.

As we know system calls run in the kernel mode and each system call has an entry point or a special “trap” instruction which when executed will result in switching to kernel mode.

Context of a process:

The context of a process is its state defined by its text or code, values of its global user variables and data structures, and values of registers used by the process, and contents of the user and kernel stacks.

When the system changes from process A to process B, it is called context switch. But **changing from user mode to kernel mode is not a context switch.**. It is only a change in mode.

Following are the different states of a UNIX process.

The process states and transition is similar to what we study in Operating Systems, but here it is more specific to a UNIX-like operating system.

1.The process is executing in user mode

2.The process is executing in kernel mode

3.The process is not executing but ready to run as soon as the kernel; schedules it

4.The process is sleeping and resides in main memory

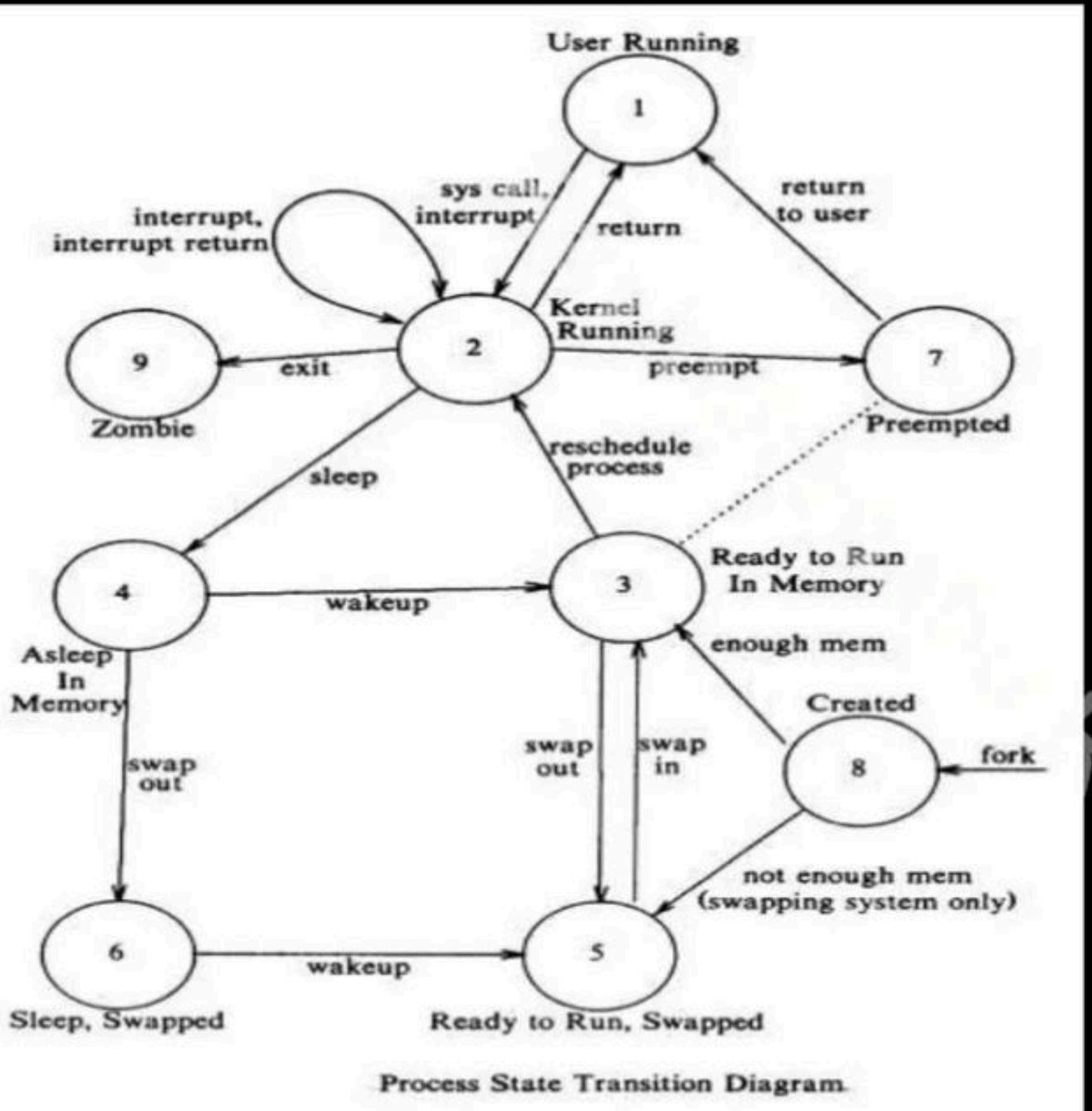
5.The process is ready to run but the swapper must swap the process into main memory before the kernel can schedule it to execute.

6.The process is sleeping, and the swapper has swapped it to secondary storage to allow other processes to run.

7.Process is returning from kernel to user mode, but the kernel preempts it and does a context switch to schedule another process.

8.The process is newly created and is in transition state.. It is not ready to run. Other than process 0 all other processes start here.

9.The process executed **exit()** system call and is in the zombie state. The process no longer exists but this state is for the parent process to collect some timing statistics. This is the final state of a process.



Process State Transition Diagram

The two data structures **'process table entry'** and the **'u area'** describe the state of a process.

Here the **process table entry** contains the fields which must be accessible to the kernel, whereas the **u area** contains fields which are accessible only to the running process.

Layout of system memory:

Each process has a virtual address space which has addresses starting from 0 onwards. These virtual addresses are mapped to physical addresses using paging which is what we study in OS as part of **“Memory management subsystem”**.
As mentioned before a process on a UNIX-like OS contains three sections: **text section**(which is the source code of the program), **data section**(has global variables) and **the stack section**(which has local variables).

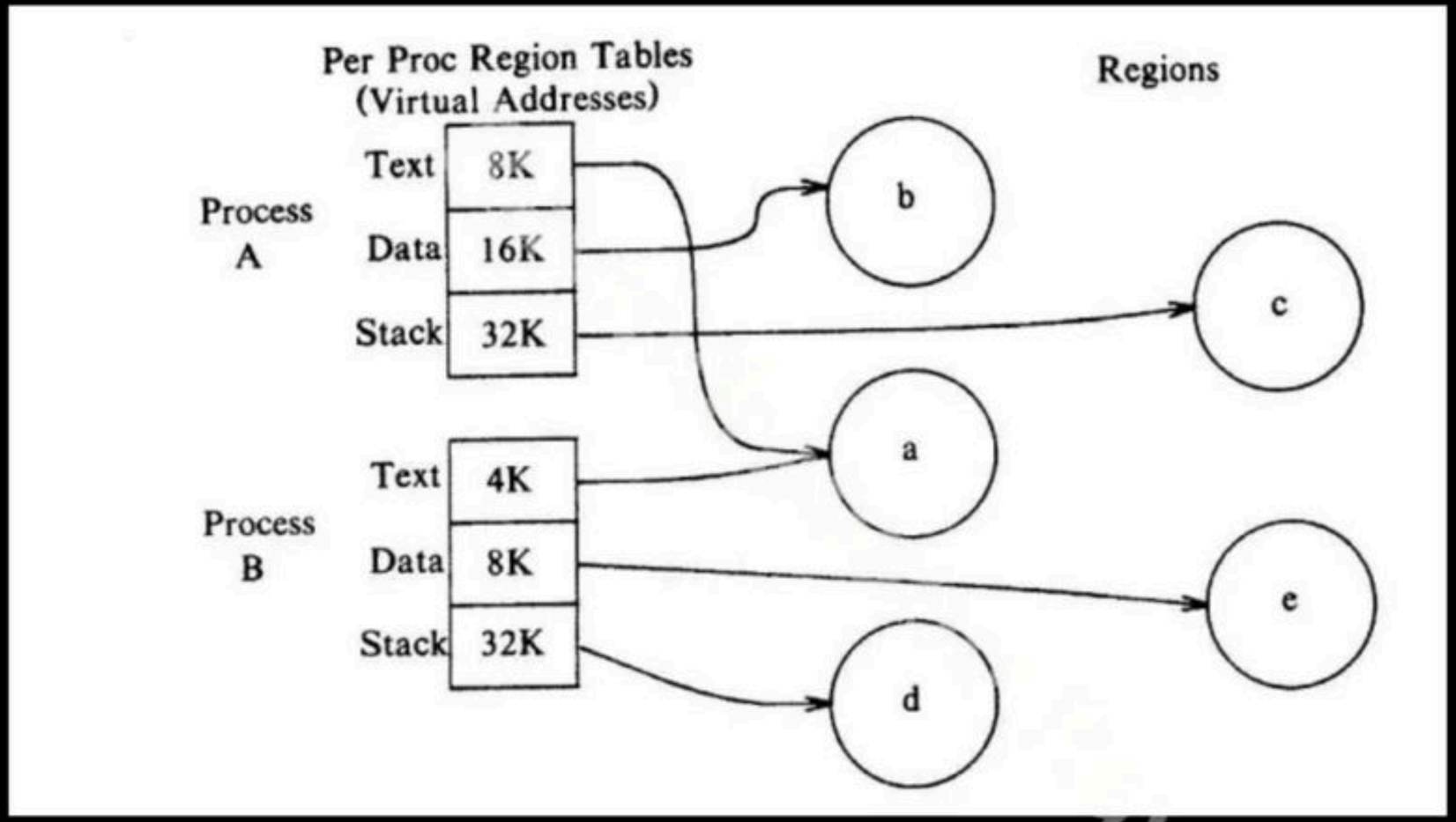
Regions:

- The system kernel divides the virtual address space of a process into separate regions, giving one region to each of the text, data and stack sections.
- We know several processes can run the same program..so they all can share the same **region** corresponding to the text section.
- The kernel maintains a region table allocating one entry for each active region in the system.
- Each process also contains a private region table, called ***pregion***.
- Each process region table entry contains the starting virtual address of the region in the process.

A region table entry also has the following other fields:

1. A pointer to the inode of the file which is loaded into the region
2. Type of the region, whether it is **text** or **data** or **stack** type.
3. Size of the region
4. Location of the region in physical memory
5. Status of the region like whether it is locked, valid, being loaded into memory etc.
6. Reference count which tells the number of processes that reference the region.

A shared region may have different virtual addresses in different processes.



As can be seen in the image, the region ‘a’ is shared between process A and process B.In process A’s context it is starting at virtual address 8K, whereas in process B’s context region ‘a’ is starting at virtual address 4K.

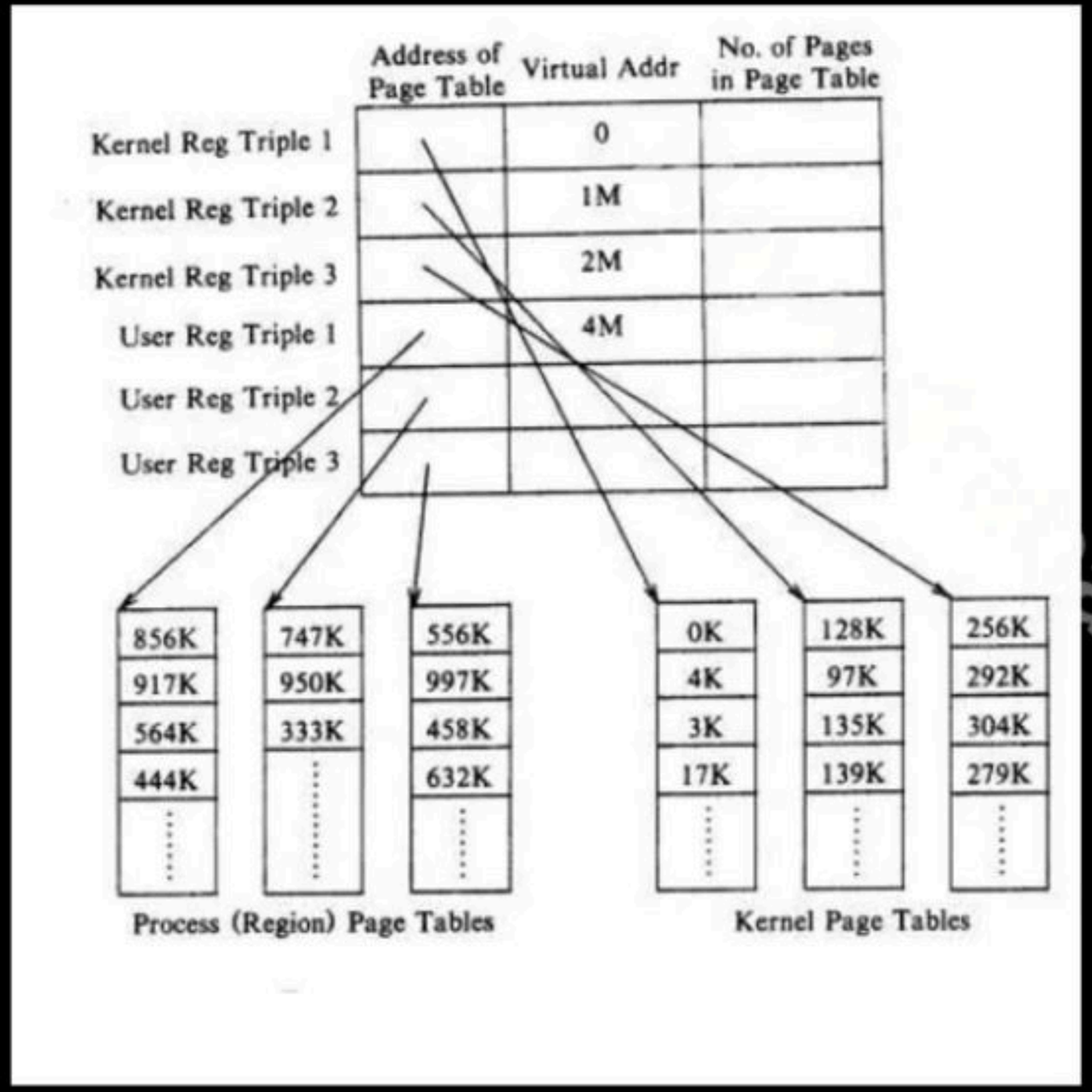
Layout of the Kernel:

The virtual memory mapping of the kernel is independent of all processes.

When the system boots it loads the kernel into the memory and sets up the necessary tables and registers to map the kernel virtual addresses into physical memory addresses.

The kernel page tables are similar to page tables associated with processes.

When executing in kernel mode the system allows access to kernel addresses and prohibits such access when running in user mode.



When changing mode from user mode to kernel mode due to an interrupt or system call..the OS collaborates with the hardware to permit kernel address references.

Manipulate the address space of a process:

Various system calls like fork(), exec(), brk() {brk system call is internally called when we run malloc() function} manipulate the virtual address space of a process.

Below are the operations that are performed by those system calls on the regions to manipulate the virtual address space:

1.Lock a region
2.Unlock a region
3.Allocate a region
4.Attach a region to the memory space of a process
5.Change the size of a region
6.Load a region from a file into the memory space of a process
7.Free a region
8.Detach a region from the memory space of a process, and duplicate the contents of a region

Locking and Unlocking a Region:

The kernel can lock and allocate a region and later unlock it, but doesn't free that region. Similarly if it wants to prevent access by other processes to an allocated region for a time, it can lock it and later unlock it.

Allocating a Region:

The kernel allocates a region out of the list of free regions when fork, exec, and shmget (shared memory get) system calls occur. When a region is to be allocated, the kernel picks up the first region from the free list and puts it on the active list. The i-node is used by the kernel to enable other processes to share the region. The kernel increments the i-node reference count to prevent other processes from removing its contents when unlinking it.

Changing the Size of a Region:

A process may expand or shrink its virtual address space with the *sbrk()* system call.
Similarly, the stack region can be changed (it can be expanded or shrunk) automatically when nesting of calls is done.
It uses the *growreg* algorithm. The *sbrk()* system call also internally calls *growreg*. Shared regions cannot be extended.

Freeing a Region:

When a kernel no longer needs a region, it frees the region and places it on the free list again.

Detaching a Region from a Process:

The kernel detaches regions by the exec, exit, and shmdt system calls. It updates the ***pregion*** entry and cancels the connection to physical memory by invalidating the associated memory management register triple.

The address translation mechanisms thus invalidated apply specifically to the process, not to the region (as in the algorithm *freereg*). The kernel decrements the region reference count. If the region referenced count drops to 0 and there is no reason to keep the region in memory, the kernel frees the region with algorithm *freereg*. Otherwise, it only releases the region and inode locks.

Duplicating a Region:

In the fork system call, the kernel requires to duplicate the regions of a process. If the region is shared, the kernel just increments the reference count of the region. If it is not shared, the kernel has to physically copy it, so it allocates a new region table entry, page table, and physical memory for the region.

My philosophy

TEACHING IS WORSHIP

STUDENTS ARE GODS