💻

Linux Course

Index

▼ Lecture 1 & 2:

Linux Concepts and Distributions:

1. **Linux Main Distributions:**
   - Red Hat
   - Debian (e.g., Kali for security, Ubuntu, Mint for educational purposes)
   - SUSE
2. **Linux Basics:**
   - Linux is a Unix-like operating system.
   - It is free and open-source software.
   - Operating system → System software → Familiar with Unix, data processing capabilities, scripting languages (e.g., AWK, Make).
   - Kernel: Primary module of the OS.
3. **GNU/Linux:**
   - GNU/Linux is an operating system, serving as a layer between hardware and user-level applications.
   - Linux: Refers specifically to the kernel.
4. **OS Roles:**
   - Hide hardware complexity and diversity.
   - Manage resources.
   - Provide isolation and protection for applications.
   - Multitasking: Ability to perform several tasks simultaneously.
5. **Main OS Features (5 A's):**
   - Help/Assistance: Provide support and guidance to users.
   - Abstraction: Hide complexity through a unified interface (API).
   - Augmentation: Improve performance, extend resources virtually, support multitasking.
   - Arbitration: Resolve conflicts, allocate resources, ensure system reliability.
   - Authorization: Protect privacy, ensure system integrity, limit access to resources for security.
6. **Linux Characteristics:**
   - Unix-like OS assembled under the model of free and open-source software.
   - Distributions cater to different purposes.

7. **WHO/WHEN:**
    - Linus Torvalds: Created Linux in 1991.
    - Richard Stallman: Developed GNU utilities in 1983.
    - In 1980, all software was proprietary.
8. **Advantages of Open Source Software:**
    - Enables global collaboration and development.
    - Lower investment costs as software belongs to the community.
9. **Statistics:**
    - 67% of servers run Linux.
    - 41.8% of known websites use Linux.
    - In 2020, 96% of the top 1 million websites were powered by Linux.
    - In 2022, 91.5% of the top 500 supercomputers use Linux.
    - 84% of enterprise businesses run Linux.
    - 58% of IoT devices run Linux, facing challenges in power and security.
    - 83% of the software industry uses Linux as the primary OS.
    - Modern smartphones and devices like Android phones, Amazon Kindle, and smart TVs use the Linux kernel, with Android being a framework built on top of it.

▼ **Why Linux:**

1. **Free and Open Source:**
    - Linux is free to use and distribute, and its source code is openly accessible, allowing for customization and collaboration.
2. **Powerful for Research Datacenters:**
    - Widely deployed in research datacenters due to its powerful capabilities, stability, and performance, making it suitable for handling complex computational tasks and large datasets.
3. **Universal:**
    - Linux is versatile and flexible, capable of running on various hardware architectures and devices, from servers and desktop computers to embedded systems and IoT devices.
4. **Personal Desktops & Phones:**
    - Linux is a popular choice for personal desktops and phones, offering a customizable and efficient operating system for everyday use.

5. **Community and Business Driven:**
    - Linux development is driven by a vibrant community of developers and organizations, collaborating to improve and evolve the operating system to meet the needs of users across diverse industries and applications.

10. **Linux Distributions:**

Linux Distribution	Year	Independence / Base	Usage	Infos
Red Hat Enterprise Linux	2000	Independent/Fedora	Servers, Enterprise	- Most commercially popular Linux Dist. Was Supported by IBM in 2019. Uses RPM/YUM for SW install&managment.
Fedora Linux	1995	Independent	Personal, Development	- Sponsored by RHEL, Same as CentOS (test-ground for RH)
OPEN SUSE Linux	1994	Independent	Servers, Enterprise	
Debian Linux	1993	Independent	Personal, Servers, Development, Stable	- Largest free SW package collection. Completly free.
Ubuntu Linux	2004	Based on Debian	Personal, Servers, Education	- Free and easy, Many OS derived from it. Funded&Supported by Canonical Ltd.
Linux Mint	2006	Based on Ubuntu	Personal, Education	- For education.

N.B: Linux distributions are customized versions of the Linux operating system, comprising packages and applications with the Linux kernel. Many distributions are based on or derived from others, allowing for diversity in features and configurations.

11. **Codecs:**

- Codecs are libraries needed for encoding & decoding videos.

12. **Classification:**

    Beginners:

    - Mint
    - Ubuntu

Beginners to Intermediate:

- Fedora
- OpenSUSE
- Mageia (Some post-installation needed)

Advanced:

- Debian
- RHEL
- Arch (Requires reading documentation and becoming comfortable with command lines)

Experts:

- Slackware
- NixOS
- Gentoo
  - Additional distro-specific knowledge required (e.g., compiling source code depending on the use, not just installation)

13. **Packet Manger:**

    The package manager in Linux is a SW tool for managing software packages installation, removal, and updates. It simplifies these tasks by automating the process, making software management easier for users. Examples include {APT (Advanced Package Tool)/ DPKG (Debian Package)} in Debian/Ubuntu, YUM (Yellowdog Updater Modified) in Red Hat/CentOS, and DNF (Dandified YUM) in Fedora.

14. **Distribution Choice Criteria:**

    - There are no strict rules for selecting a specific Linux distribution.
    - Choose based on requirements, skills, and budget.

15. **Recommendation:**

    a. Red Hat Derived Distributions:

       - Recommended for Enterprise (Large) size networks.
       - Examples include CentOS and Fedora.

    b. Ubuntu Derived Distributions:

       - Recommended for Small/Midsize networks and personal use.
       - Examples include Linux Mint and Elementary OS.

▼ Lecture 3:

1. **GNU and Free Software Foundation (FSF):**
    - GNU stands for "GNU's Not Unix," aiming to create a complete free software system compatible with Unix.
    - FSF was founded in October 1985 by Richard M. Stallman to develop and raise funds for GNU software.
2. **Main GNU Software:**
    - Bash (Shell)
    - GCC (C compiler)
    - GDB (Debugger)
    - GIMP (Image Manipulator)
    - Gnome (Desktop Environment)
    - Emacs (Text Editor)
    - Ghostscript and Ghostview (.ps files interpreter and graphical frontend)
    - GNU Photo (Digital Cameras)
    - G++ (C++ Compiler)
    - Octave (Image Processing)
    - GNU SQL
    - Radius.
3. **Open Source and Licenses:**
    - Open source software: In Richard Stallman POV; “Free-Software”. grants users the freedom to run, copy, distribute, change, and improve it. (This is Theoric Part)
    - Software licenses (Pratical Part), like the GNU General Public License (GPL), govern the use or redistribution of software.
    - GPL ensures and governs freedom for users and requires modified source code to be passed along, and not its use.
    - Copyleft requires any distribution to be published under *the same name* and conditions.
    - Open source licenses include GNU GPLv3 {contaminating} , GNU LGPLv3 {not attached}, Mozilla PL 2.0, MIT License.

- License choice is a decision of the legal department and can be determined by *reverse engineering binary code.*

4. **Shell:**
    - **Definition:** The shell is a software component and a main part of Linux.
    - **Interface:** It acts as an interface between the user and the kernel (OS).

- **Characteristics:**
    - Command Line Interface (CLI): Not graphical; commands are typed using the keyboard, without mouse interaction.
    - Examples: Bash (Bourne Again Shell), Zsh (Z Shell), Ksh (Korn Shell).
- **Bash (Bourne Again Shell):**
    - Developed as the GNU version in 1977.
    - Named after its author, Stephen Bourne, an English computer scientist.
    - Derived from the original Unix shell, 'sh.'
    - Contains the most features and is the default shell in many Linux distributions.
- **Functionality:**
    - Interprets commands and sends them to the operating system.
    - Provides built-in commands, programming control structures, and environment variables.
    - Supports scripting with commands **interpreted** rather than compiled.
- **Environment Variables:**
    - Pre-defined variables that can be manipulated within a script.
- **Multiple Shells:**
    - Linux supports various shells such as Bash, csh, zsh, ksh.
    - MacOS defaults to zsh but can install other shells.

Understanding Bash allows users to adapt to other shells easily due to their similarities. Bash checks syntax errors and executes commands once it confirms their correctness.

5. Command:

- **Definition:** A command is a software program that performs a specific task when executed in the shell.
- **Creation:** Users can create their own commands by writing code and calling it in the shell.

- **Aliases:** Users can customize existing commands by creating aliases, which are alternative names for commands with specific options.
    - alias nickname = “command”
- **Prompt (Green Part in Mint):**
    - **Definition:** The prompt is the part of the command line interface that indicates the system is waiting for commands.
- **Components:**
    - **Username:** Username of the current user.
    - **Systemname:** Name of the system.
    - **Currentdirectory:** Current working directory.
        - **~:** Represents the home directory.
    - **$:** Represents a normal (regular) user.
        - **#:** Represents the superuser.

    ```
    [username@systemname ~]$|
    ```

▼ **Input:**

- **Format:** `command --option argument.`
- **Components:**
    - **Command:** A program that performs one specific task.
    - **Options:** Modify the behavior of the command.
        - Short Form: Single dash followed by one letter (e.g., ls -a).
        - Long Form: Double dash followed by a word (e.g., ls --all).
    - **Argument (Parameters):** Input/output that the command interacts with.
- **Execution:**
    - Commands are small programs that perform one task well.
    - Root privileges can be attained without switching users using "sudo command", which is preferred.
    - UNIX philosophy emphasizes the power of system relationships among programs rather than individual programs.
    - Combining multiple commands can create powerful and useful tools.
        - `;` : to type multiple commands on one line.
        - `\` : to split the commands accross multiple lines.

▼ Lecture 4:

1. **Commands:**
    - `whoami` : Display current user; name of login.
    - `hostname` : Display system hostname; name of machine. (You can open many terminals in other machines)
    - `echo "string"` : Print specified string.
        - `echo -n` : dosent output a newline at the end. (i.e: no endl)
    - `echo $HOME` : Print user's home directory path; considered as an ENVIRONMENT VARIABLE. $ sign to access its content.
        - `export` : To make a shell environment variable.
            - `export VAR_NAME = “*”`
        - `env` : to view your environment variables.
    - `echo my login is $(whoami)` : Print a string then replace whoami with its value (user's login name.)
        - Without parenthesis : is used to execute a command and substitute its output into a variable or another command. (here `whoami` is considered a command, not a variable)
        - With parenthesis : Execute the command and print the output in same line
    - `date` : Display current date and time.
    - `cal` : Display calendar.
    - `google` : Bad command.
    - `ps` : display current running processes (running in the current terminal session), snapshot in the command moment.
        - `ps aux` : a detailed list of all processes running on the system.
    - `top` : display current running processes, in real time.
    - `ls` : list files.
        - `ls [A-Z]*` : listing files starting with capital letters, + anything.
    - `cd` : change directory.
        - `cd ..` : change to parent directory.
        - `cd` `/` `cd ~` : change current directory to home.
2. **Get Help:**

▼ Tools for learning about commands, e.g., working with `date`:

- `type date`: Show type of command. (Shell built-in, Alias, Path of a file/sw)
- `date --help`: “option” help, provided by the developper. (-h abbr)
- `help date`: “command” help, for shell built-in commands only.
- `man date`: Manual for `date` command, it uses the `less` command.
    - For most executable programs intended for commandline use provide a formal piece of documentation called a manual or a man page. A special paging.
    - They do not usually include examples.
- `apropos date`: List commands related to `date` , used with ***keywords***; it basically search the manual pages for ***matches.***
- `whatis date`: Brief description of `date` , in 1 line.
- `info date`: Provided by GNU, Alternative documentation to `man` for `date`. They are hyperlinked.
    - N.B: `less` is similar to `cat` .
        - `less` : a command that allows reading text files

3. **Ways of scrolling in `less`:**
    - `space`, `f`: Page forward.
    - `b`: Page backward.
    - `<`: Go to first line of file.
    - `>`: Go to last line of file.
    - `/`: Search forward (`n` to repeat).
    - `?`: Search backward (`N` to repeat).
    - `h`: Display help.
    - `q`: Quit help.

    It's important to remember essential options for commands like `less`, but you don't need to memorize all options.

4. **Scripts:**
    a. Definition: A script is a set of commands assembled in a specific way, functioning like a program.., it recieves user commands and executes them.

    b. Example:

```
#!/bin/bash

if [ "$1" == "h" ]; then
```

```
    echo "Hello"
fi

if [ "$1" == "b" ]; then
    echo "Bye"
fi
```

`#!/bin/bash` at the beginning of a script indicates that the script should be interpreted and executed using the Bash shell.

> `#!` : shebang/hashbang

- `“$n”` : to access *nth* argument passed to the script.

▼ **Execution:**

- To execute the script: `./test.sh` in a bash shell.
- To pass a parameter to a script: `./test.sh h`.
    - The script prints "Hello" if "h" is specified as an argument.
    - The script prints "Bye" if "b" is specified as an argument.

A script is executed by invoking its file name preceded by `./` in a bash shell. Parameters can be passed to the script, allowing it to perform specific actions based on those parameters.

5. **How to create manual pages:**

    To create a manual page:

    1. Determine the type of manual page needed (e.g., general commands, system calls, library functions).
    2. Use the roff markup language (troff, nroff are variants) to write the manual page.
    3. Include commands (read markers) for various titles and sections:
        - `TH`: Title heading (should be the first command).
        - `SH`: Section heading.
        - `B`: Bold text.
        - `TP`: Information about an argument (flag) to the command.
        - `BR`: Text in bold and normal Roman font.

    Convention for Man Pages:

    - Man pages are created for various types of information:

    1. General commands manual

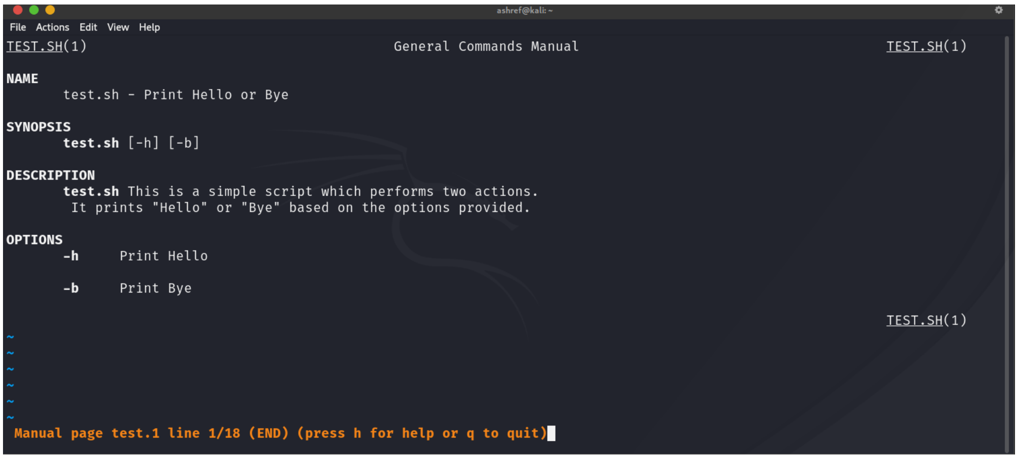
2. System calls manual
3. Library functions manual
4. Kernel interfaces manual
5. File formats manual
6. Games manual
7. Miscellaneous Information manual
8. System manager's manual
9. Kernel developer's manual

These conventions help standardize the creation and organization of manual pages, making them easier to navigate and understand.

▼ **HOW:**

1. **Create a Text File** `test.1`**:**

```
.TH TEST.SH 1
.SH NAME
test.sh \- Print Hello or Bye

.SH SYNOPSIS
.B test.sh
[\-h]
[\-b]

.SH DESCRIPTION
.B test.sh
This is a simple script which performs two actions.
 It prints "Hello" or "Bye" based on the options provided.

.SH OPTIONS

.TP
.BR \-h
Print Hello

.TP
.BR \-b
Print Bye
```



```
TEST.SH(1)                        General Commands Manual                         TEST.SH(1)

NAME
       test.sh - Print Hello or Bye

SYNOPSIS
       test.sh [-h] [-b]

DESCRIPTION
       test.sh This is a simple script which performs two actions.
        It prints "Hello" or "Bye" based on the options provided.

OPTIONS
       -h      Print Hello

       -b      Print Bye

                                                                                  TEST.SH(1)
~
~
~
~
~
~
 Manual page test.1 line 1/18 (END) (press h for help or q to quit)
```

- For more information about manual page syntax, refer to the manual page of the manual (`man man`).
    - **Storage of Man Pages:**
        - The system stores its man pages in `/usr/share/man/` .
        - **The directory** `/man/man1` specifically stores man pages for user shell commands.

⇒ Files with the **.gz** extension are compressed using the ZIP compression format. To read them without decompressing, we can use the `zcat` command.

⇒ It is recommended to store your own man pages in `/usr/` `local` `/man` to avoid conflicts with system-managed man pages located in `/usr/` `share` `/man` .

▼ Lecture 5:

N.B:

- `man -k ...` is equivalent to `apropos ...` ; keywords.
- `man -f ...` is equivalent to `whatis ...` .

  If these commands do not behave as expected, it could be due to:

    - Different shell settings.
    - Different versions of `man` or `apropos` .
    - Presence of aliases.
    - Variations in the content of man files.

⇒ Linux is highly customizable and modifications are safer compared to other systems. However, becoming a root user in Linux requires experience, and there are certifications available to become a Linux administrator.

1. **User Management:**

   User management ensures:

   - Secure access control.
   - Resource allocation.
   - System administration.

   ⇒ Each user is associated with a user account, defining their identity and privileges. Users can have different privileges, and multiple users can connect to the same machine, even remotely.

2. **Types of Users:**

   a. **Root Account (Superuser):**

      - Has complete control of the system.
      - Can run any command without restrictions.
      - Should be treated as a system administrator.

   b. **System Accounts:**

      - Created by the system during installation.
      - Used to run system services and applications.
      - Modifications to these accounts could affect the system adversely.

   c. **User Accounts:**

      - Created by the administrator.
      - Access the system and its resources based on permissions.
      - Provide interactive access to the system for users and groups.
      - Generally have limited access to system files and directories.

   ▼ **Additional Information:**

      - Root account (# in the prompt) has all permissions.
        - But he can't know the passwords, for privacy; they are encrypted using `SHA`
          - if you lost your password, you can't recover it, but the root can create another password.
      - You can have many root account, but with different names.
      - System creates accounts when processes are launched.
      - Killing a process or deleting an account can provide insights into the purpose of system accounts.
      - System accounts, privileges, etc., can be displayed and managed.

  - Linux is designed to manage different users efficiently.
  - Each user belongs to one group at least.

 Challenges Faced:

  - Security.
  - Resource management.
  - Permissions management (Privacy).
  - Interrupt handling.
  - Use of groups to manage privileges efficiently.

3. **Group user:**
   1. **Definition:**
      - Groups are collections of users, simplifying management of multiple users, particularly regarding permissions.
   2. **Permissions:**
      - Assigned to groups of users with identical permissions, organized into logical groups.
      - Users in a group share the same permissions.
   3. **Management:**
      - Groups can be modified, and users can be moved between groups.
      - Users can belong to multiple groups, with their permissions being the combination of all group permissions.
   4. **Administrative Efficiency:**
      - Admins can manage permissions for entire user groups, streamlining permission management instead of handling individual user accounts.
4. **User properties:**
   1. **Username:**
      - Cannot start with a number or include spaces.
      - Policy defined by a regular expression.
   2. **UID (User ID):**
      - First assigned as 1000, increments.
      - IDs < 1000 are system accounts.
      - 0 is root ID.

        - **UID -1 or 4294967295**: This is an invalid UID, often used to indicate no user.

    c. **GIDs (Group IDs):**

    - Users can belong to different groups.
    - Similar characteristics to UID
        - 0 is root GID, and so on...

    d. **Home Directory:**

    - Default: `/home/username/`.
    - Customizable.

    e. **Default Shell:**

    - Shell can be changed.

    f. **Password:**

    - Protects account to prevent unauthorized access.

5. **How user managment works:**

    a. **Data Storage:**

    - Linux stores user and group data in specific files and directories.
    - These files/directories contain:
        - user account infos.
        - encrypted passwords.
        - group configurations.

    b. **File Contents:**

    - `/etc/passwd`: List of user accounts and corresponding info. Readable by most users; only root and sudo accounts can modify. Also stores installed packages.

      ```
      ashref:x:1000:1000::/home:/bin/bash
      ```

    - `/etc/group`: List of user groups, displaying group name, GID, and members.
    - `/etc/sudoers`: Specifies users with elevated permissions (sudo command usage).
    - `/etc/shadow`: Stores encrypted user password info and related data.
    - `/etc/gshadow`: Stores encrypted group password info and related data.
    - `/etc/skel`: Contains default configuration scripts and templates copied to a new user's home directory.

- `/etc/login.defs`: Contains system-wide user account policy settings.

c. **Modification:**

- System administrators interact with these files to control and modify user/group settings.
- Users initially modified these files for permission changes, contributing to the expertise of the root user.
- Risk is involved in modifying these files, so caution is necessary.

⇒ Understanding the structure and contents of these files/directories is essential for effective user management in Linux.

6. **The commands to use:**

a. **id**: Display user and groups IDs.

- Example: `id usrnm`
- `id -nG test_account` **:** The **-n** and **-G** options instruct `id` to only list group names instead of *numeric IDs*.

b. **finger**: Display detailed user information.

- Example: `finger usrnm`

c. **lslogins**: Display user information in Linux.

- Example: `lslogins -u`

d. **groups**: List user group membership.

- Example: `groups usrnm`

e. **getent**: Fetch user information from system database.

- Example: `getent passwd usrnm`

f. **grep**: Search for patterns or specific text in files.

- Example: `grep usrnm /etc/passwd`
  - `grep -i` : case insensitive. / `[xy]` character class.

g. **users**: List currently logged-in users on Linux.

- Example: `users`

h. **who**: Show information of currently logged-in users.

- Example: `who -u`

i. **cat**: List all users from `/etc/passwd`.

- Example: `cat /etc/passwd`

j. **last**: Show most recent login session.

- Example: `last`

k. **lastb**: Show failed login attempts.

- Example: `lastb`

▼ More managment:

Create User:

- `useradd`: Command to create a new user in Linux, requires root or sudo privileges.
  - Example: `sudo useradd test_account`
  - No additional information needed.
- `adduser`: *Interactive* command to create a new user.
  - Automatically creates a home directory, sets a default shell, and prompts for a password.
  - Example: `sudo adduser test_account`
  - Check creation by verifying `/etc/passwd` (using `cat`, `sed`, `awk` or `grep`).

Modify Default User Settings:

- `usermod`: Modifies various attributes of an existing user account.
  - Options:
    - `-d`: Change user's home directory.
    - `-s`: Change user's shell.
    - `-e`: Set expiry date.
    - `-c`: Add comments to user entry.
    - `-u`: Change user's ID.
    - `-aG`: Add user to supplementary groups without removing existing group membership.
    - `-G`: remove the user from a group.
  - Example: `sudo usermod -d /var/acc2 acc2` (changes directory of user named acc2 to /var/acc2).

Delete User:

- `userdel`: Removes a user from the `/etc/passwd` file.
  - Example: `sudo userdel test_account`
  - To remove all related files from system: `sudo userdel -r test_account`

Group Management:

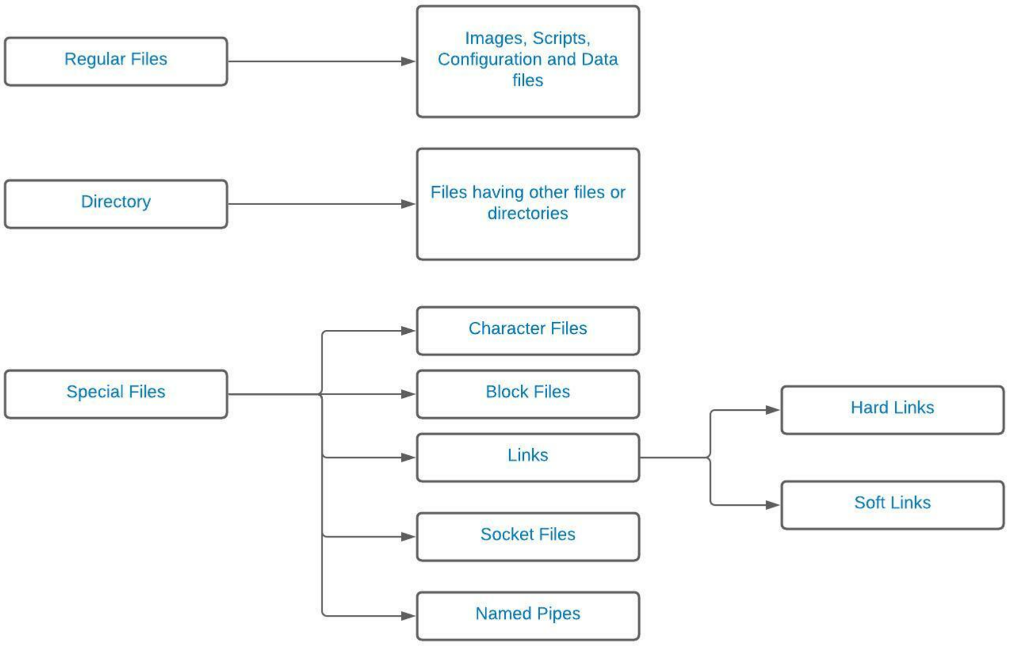
- `addgroup`: Command to create a new group.
- `groupdel`: Command to delete a group.
- Add User to Group: `sudo adduser test_acc test_group`
- Remove User from Group: `sudo deluser test_acc test_grp`

These commands enable the creation, modification, and deletion of users and groups in a Linux system.

▼ Lecture 6:

Linux File System Structure

1. **Introduction to Linux File System**:
    - Unix philosophy: "Everything is a file; if something isn't a file, it's a process."
    - Concept of data processing in Computer Science.
2. **Organization of Data in Linux**:
    - Data in Linux is organized in files.
    - Files are organized into directories.
    - Directories are organized into a tree-like structure.
3. **Unified Treatment of Files and Directories**:
    - Linux, like Unix, makes no distinction between a file and a directory.
    - All types of data are treated as files in Linux.
    - Generally, all I/O devices are represented as files in Linux.
4. **Types of Files**:
    - **Regular(Ordinary) Files**: Contain data, text, program instructions, etc.
    - **Directories**: Store both special and ordinary files. Equivalent to folders in Windows/MacOS.
    - **Special Files**: Provide access to hardware such as hard drives, CDs, modems, Ethernet adapters, etc.
        - **Links (Aliases or Shortcuts)**: Enable accessing a single file using different names.



5. **Listing Content of a Directory**:
    - Command `ls` is used to list the content of a directory.
    - The `ls` command supports the `l` option to get more information about the listed files.
    - Additional information provided by the `l` option includes:
    - **The Honor (Owner)**: The person who created the file. (Not always accurate if the creator belonged to another user).
        - An example of the output of `ls -l` with all the details of the numbers:

```
total 64
-rw-r--r-- 2 user group 4096 Mar  4 15:32 example.txt
```

In this example:

1. **File Type and Permissions**:
    - `rw-r--r--`: This represents the permissions of the file. The **first** character `-` indicates that it's a regular file. The following nine characters (`rw-r--r--`) represent the permissions for the owner, group, and others. In this example:
        - `rw-`: Owner (`user`) has read and write permissions.
        - `r--`: Group (`group`) has read-only permission.
        - `r--`: Others have read-only permission.
            - The similarities between these permissions:

    1. **Read (`r`) Permission**:

        - This permission allows users to view the contents of the file. For a directory, it allows users to list the files within it.

    2. **Write (`w`) Permission**:

        - This permission allows users to **modify** the contents of the file. For a directory, it allows users to create, delete, or rename files within it.

    3. **Execute (`x`) Permission**:

        - This permission allows users to execute the file as a program or script. For a directory, it allows users to access files and subdirectories within it.

    In summary, "rw-r--r--" means:

    - The owner of the file has read and write permissions.
    - The group associated with the file has read-only permission.
    - Others (users not in the owner group) have read-only permission.

> Directory permissions with `t` at the end; only the owner can delete or rename files within that directory; eventhough the directory is global writable. (Sticky Bit)
>
> e.g.: `drwxrwxrwt`

2. **Number of Links**:

    - `2` : This indicates the number of hard links to the file. In addition to the original file itself, there is one more hard link pointing to the same inode.
        - You can find `@` indicating soft links, or `|` indicating named pipes. at the end.

3. **Owner and Group**:

    - `user` : This is the owner of the file.
    - `group` : This is the group associated with the file (only one).

4. **File Size**:

    - `4096` : This represents the size of the file in bytes. For directories, this number represents the size occupied by the directory entry itself, not the size of its contents.

    5. **Timestamp**:
        - `Mar 4 15:32`: This represents the date and time the file was last modified.
    6. **File Name**:
        - `example.txt`: This is the name of the file.
    7. **Total**:
        - the total block count is 64, indicating that the listed files and directories collectively use 64 blocks of storage on the disk.

6. **File types:**

    File types are represented by characters at the beginning of the output when you list files using commands like `ls -l`. Here are the common file types and their representations, FOUND AT THE BEGGINING when typing ls -l:

    1. **Regular(Ordinary) file (-)**:
        - A regular file contains data or text. It can be a document, script, binary executable, etc.
        - Represented by the hyphen (-) character.
    2. **Directory (d)**:
        - A directory is a special type of file that contains other files and directories.
        - Represented by the letter "d".
    3. **Special Files:**

        a. **Symbolic link (l)**:
        - A symbolic link, also known as a symlink or *soft* link, is a special type of file that points to another file or directory.
        - Represented by the letter "l".

        b. **Block special file (b)**:
        - A block special file represents a device that is accessed as a sequence of blocks or chunks of data.
        - Represented by the letter "b".

        c. **Character special file (c)**:
        - A character special file represents a device that is accessed as a stream of bytes or characters.
        - Represented by the letter "c".

        d. **Named pipe (FIFO) (p)**:

- A named pipe, also known as a FIFO (first in, first out), is a special type of file used for inter-process communication.
- Represented by the letter "p".

e. **Socket (s)**:

- A socket is a special type of file used for inter-process communication between processes on the same or different hosts.
- Represented by the letter "s".

These file types provide information about the nature of the file and how it should be interpreted or accessed by the operating system and applications. When you use commands like `ls -l`, the first character in the file listing indicates the type of file.

7. **Metacharcter**:

- **Metacharacters**: Metacharacters are special characters in Unix-like systems with predefined meanings.

1. **Wildcards:**
    - Wildcards represent one or more characters in file and directory names.
    - Examples:
        - `*` - Matches zero or more characters.
        - `?` - Matches any single character.
        - `[?-?]` - Matches any character within the specified range or set.
            - **Usage**:
                - `*`: Matches zero or more characters.
                    - Example: `file*` matches `file1`, `fileA`, `fileABC`, etc.
                - `?`: Matches a single character.
                    - Example: `file?` matches `file1`, `fileA`, but not `file10` or `fileABC`.
                - `[-]`:
                    - Example: `ls file[0-9].txt` matches files like `file1.txt`, `file2.txt`, etc... until `file9.txt` .
2. **Redirection and Pipes:**
    - Redirection and pipes are used to control input, output, and error streams, and to chain commands together.
    - Examples:
        - `>` - Redirects output to a file, overwriting its contents.

        - `>>` - Redirects output to a file, appending to its contents.
        - `<` - Redirects input from a file.
        - `|` - Sends the **output** of one command as **input** to another command (pipe).

    3. **Escape Characters:**

        - Escape characters are used to remove the special meaning of meta characters.
        - Examples:
            - `\` - Escapes the following character, treating it as a literal character rather than a meta character.

8. **Hidden Files:**

    - **Identification**: Hidden files in Unix-like systems start with a dot (.) character in their filenames.
    - **Visibility**: They are not displayed in directory listings by default, but can be shown using the `a` option with the `ls` command (`ls -a`).
    - **Purpose**: Hidden files are commonly used for ***configuration files*** or settings that are not meant to be directly manipulated by users.
    - **Example**: An example of a hidden file is `.bashrc`, which contains configurations for the Bash shell.

9. **Permissions**:

    In Unix-like operating systems, every file has three sets of permissions:

    - Owner permissions: Determine what actions the owner of the file can perform (read, write, execute).
    - Group permissions: Determine what actions members of the group associated with the file can perform.
    - Other (world) permissions: Determine what actions users who are not the owner or part of the group can perform.

        let's consider a file named "example.txt" with the following permissions:

        ```
        -rw-r--r--
        ```

        In this example:

        - Owner permissions (rw-): The owner of the file has read and write permissions.
        - Group permissions (r--): The group associated with the file has read-only permission.

- Other permissions (r--): Users who are not the owner nor part of the group have read-only permission.

▼ To change the file and the directory permissions:

The `chmod` command is used. There are two primary methods to modify permissions with `chmod`:

1. **Symbolic Mode**: Uses symbols (+, -, =) to add, remove, or set permissions for the owner, group, and others.
2. **Absolute Mode**: Uses numeric values (0-7) to explicitly set permissions for the owner, group, and others.

   a. **Symbolic Mode Example**:
   To add execute permission for the owner, we use the command:

   ```
   chmod u+x, g=rx, o-wx example.txt
   ```

   - `u+x`: Adds execute permission for the owner.
   - `g=rx`: Sets read and execute permissions for the group, while removing any other permissions.
   - `o-wx`: Removes write and execute permissions for others.

   So, applying this command to the file "example.txt" would result in the following permissions changes:

   - Owner permissions: Execute permission added.
   - Group permissions: Read and execute permissions set.
   - Other permissions: Write and execute permissions removed.

   After running the command `chmod u+x, g=rx, o-wx example.txt`, the permissions of "example.txt" would be modified accordingly.

   - The output of `ls -l`:

   ```
   -rwxr-xr-- 1 <owner> <group> <date> example.txt
   ```

   b. **Absolute Mode Example**:
   To set read and write permissions for the owner, and read-only permissions for the group and others, we use the command:

   ```
   chmod 743 example.txt
   ```

   - `7` specifies permissions for the owner (user).

    - The binary representation of 7 is 111, indicating read (4) + write (2) + execute (1) permissions.
    - So, the owner (user) gets read, write, and execute permissions.
- `4` specifies permissions for the group.
    - The binary representation of 4 is 100, indicating only read (4) permission.
    - So, the group gets read-only permission.
- `3` specifies permissions for others.
    - The binary representation of 3 is 011, indicating write (2) + execute (1) permissions.
    - So, others get write and execute permissions.
        - The output of `ls -l`:

        ```
        -rwxr--wx 1 <owner> <group> <date> example.txt
        ```

        ▼ **First Digit (Owner/User)**:

        - Represents the permissions for the owner (user) of the file.
        - Each digit can be a combination of 0, 1, 2, 4, or their sum:
            - 0: No permissions.
            - 1: Execute permission.
            - 2: Write permission.
            - 4: Read permission.
        - Example:
            - 0: No permissions.
            - 1: Execute permission.
            - 2: Write permission.
            - 3: Execute and write permissions.
            - 4: Read permission.
            - 5: Read and execute permissions.
            - 6: Read and write permissions.

        - 7: Read, write, and execute permissions.

    - **Second Digit (Group)**:
        - Represents the permissions for the group associated with the file.
        - Same values as the first digit, representing execute (1), write (2), and read (4) permissions.
    - **Third Digit (Others/World)**:
        - Represents the permissions for users who are not the owner nor part of the group associated with the file.
        - Same values as the first digit, representing execute (1), write (2), and read (4) permissions.

    OWNERSHIP:

    - `chown`: Changes the owner of a file or directory.
    - `chgrp`: Changes the group of a file or directory.

    These commands are used in Unix-like systems to manage ownership and group ownership of files and directories

10. **Linux file system Layout:**
    a. The Linux file system layout can be visualized as a tree structure, with the root directory (/) serving as the **trunk** from which all other directories and files branch out. Here's a brief overview of the layout and its key components:

    1. Root Directory (/):

    - The starting point of the file system tree, indicated by a forward slash (/).
    - All other directories and files are organized beneath it.
    - Contains essential system **directories** and files, including:
        - `/bin`: Essential executable binaries.
        - `/etc`: Configuration files for system-wide settings.
        - `/home`: User home directories; Primary hierarchy for user files.
        - `/lib` & `/lib64`: Shared libraries required by system binaries.
        - `/usr`: Secondary hierarchy for user-related files and programs.
        - `/dev`: Device files representing hardware devices.
            - `/dev/null`: a file that discards erros.
        - `/boot`: Essential files for booting the system.

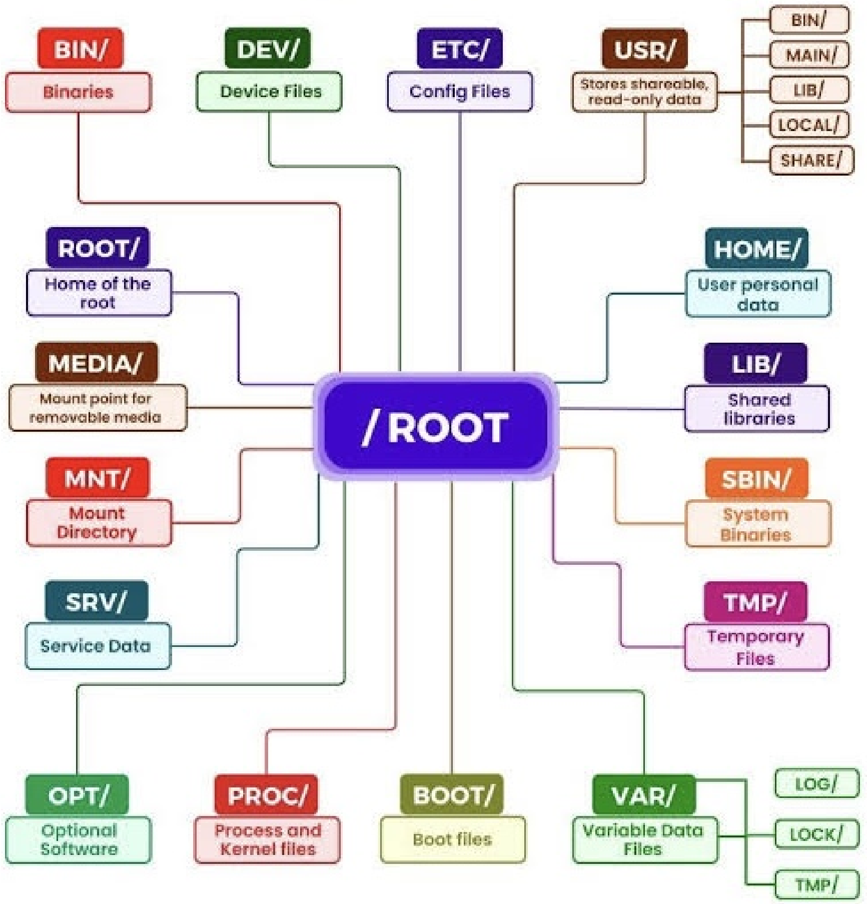
    - `/lost+found`: Recovery directory for filesystem checks.
    - `/mnt`: Common mount point for external filesystems.
    - `/var`: Stores log files, variables...

2. **Root Directory vs. Root User Home Directory**:

- While the root directory (/) is the top-level directory in the file system tree, the root user's home directory is located at `/root`.
- The root user, also known as the superuser or administrator, has full access and control over the entire **file system.**

3. **Tree Structure**:

- The file system layout follows a hierarchical tree structure, with directories containing subdirectories and files.
- Each directory serves as a branch in the tree, containing files and subdirectories as its leaves.



/dev Directory:

- Contains references to all CPU peripheral devices hardware.
- Represented as files with special properties.
- Essential for the OS and cannot be removed.
- Contains device files corresponding to physical devices or system components.
- Device files serve as interfaces to device drivers.

/media Directory:

- Standard mount point for removable media like CDs and floppies.

/mnt Directory:

- Standard mount point.

▼ Lecture 7:

File System Reality - Partitions:

1. **Storage Organization:**
   - Linux organizes storage as a set of partitions on the same disk.
   - Multiple partitions are used for higher data security in disaster scenarios.
   - Partitioning allows the disk to be treated as independent storage areas.
2. **Benefits of Partitions:**
   a. **Data Security:**
      - Partitioning enhances security by isolating data.
      - It reduces the risk of catastrophic loss during disasters.

   b. **Independent Storage Areas:**
      - Partitions create distinct storage spaces.
      - This improves data organization and management.

   c. **Easier Backups:**
      - Partitioned disks facilitate individual backup and restoration processes.
      - This simplifies data management.

   d. **Problem Isolation:**
      - Partitions confine issues to specific areas.
      - It aids in troubleshooting and system maintenance.

e. **Avoiding Interference (Many OSs):**

   - Partitioning prevents conflicts between multiple operating systems.
   - It allocates dedicated spaces for each OS's files.

3. **Types of Partitions on Linux:**

   a. **Data Partition:**

   - Contains normal Linux system data.
   - Includes the root partition necessary for system startup and operation.

   b. **Swap Partition:**

   - Acts as an extension of the computer's physical memory.
   - Provides additional memory space on the hard disk.

4. **Partition Management:**

   - **Setting Partition Type:**
     - Tools like `fdisk` are used to set the partition type during system installation, view and manage it.
   - **Division of Hard Disks:**
     - Determined by the system administrator based on system requirements and usage scenarios.
   - **More:**
     - `df` :displays information about disk space usage on *mounted filesystems*. It shows the amount of space used, available, and total space on each filesystem.
     - `du` : provides disk space usage information f*or files and directories.*

5. **Pages:**

   - **Definition:**
     - Pages are fixed-size blocks of memory used for virtual memory management.
   - **Purpose:**
     - Divides virtual memory into uniform units, facilitating efficient memory allocation and storage.
   - **Key Functions:**
     - Enables address translation, memory allocation, and handling of page faults.
     - Optimizes memory usage and system performance

6. **INODE:**

   Inode: A serial number containing information about the actual data comprising the file, its ownership, and its location.

   - Contains metadata about the file, such as its permissions, timestamps, and size.
   - Stores the physical location of the file data on the disk.
   - Each file is associated with one inode.
   - Created during disk initialization, with *a fixed number of inodes per partition*, determining *the maximum number of files that can exist on the partition*.
   - Typically, there's 1 inode per 2 to 8 KB of storage.

   ▼ Inode Table example:

Inode Number	File/Directory Name	Metadata	Location
1	file1.txt	Permissions: rw-r--r-- Size: 1024 bytes	Disk Block: 1024-2047
2	dir1	Permissions: rwxr-xr-x Size: 4096 bytes	Disk Block: 2048-6143
3	file2.jpg	Permissions: rw-rw-r-- Size: 2048 bytes	Disk Block: 6144-8191
4	file3.docx	Permissions: rw-rw---- Size: 3072 bytes	Disk Block: 8192-11263
5	dir2	Permissions: rwxr-xr-x Size: 8192 bytes	Disk Block: 11264-19455

7. **Inode Content:**

   - File permissions
   - Number of links to that file
   - Owner user and group
   - File size
   - File type (regular, directory)
   - Last modification time
   - File or directory name
   - Date and time of creation, last read and change
   - An address defining the actual location of the file

   Additional Details:

   - Inodes contain all file metadata except for file names and directories.

- File names and directories are stored in special directory files.
- The system correlates file names with inode numbers to create a user-friendly tree structure.
- Inode numbers (ID not number of inodes) can be displayed using the `ls -i` command.
- Inodes have dedicated space on the disk.

Example Output:

```
ashref@ash:~$ ls -i
    1234 file1.txt
    5678 file2.txt
    91011 file3.txt
```

▼ **RESULT**

The `ls -il` command provides detailed information about files including their inode numbers. Here's an example output:

```
ashref@ash:~$ ls -il
total 248
1234 -rw-r--r-- 1 user user 1024 Jan 1 12:00 file1.txt
5678 -rw-r--r-- 1 user user 2048 Jan 1 12:00 file2.txt
91011 -rw-r--r-- 1 user user 4096 Jan 1 12:00 file3.txt
```

8. **Inode Uniqueness:**
   Files in different partitions can have the same inode number, other than that, each inode has a specific number (ID)
9. **Filesystem Identification:**
   ⇒ Use the
   `df filename` command to determine the partition where a file is stored.
10. **THE PATH:**
    a. **Absolute Path:**
    Full path starting from the root directory. Make no assumptions about the current working directory. Begins with the root directory
    `/`. Example: `/usr/share/aclocal/pkg.m4`

    b. **Relative Path:**
    Path relative to the current working directory. Specifies the location of a file or directory in relation to the current working directory. Example:
    `../dir/file.txt`

11. Linking Files:
A link is a way of associating multiple file names with the same set of file data, creating a shortcut or alias to access the same file content using different names. There are two types of links:

- **Hard Links:**
  Directly point to the same inode and share the same file data.
- **Symbolic Links (Soft Links):**
  Pointers to the file name and can cross file system boundaries. Symbolic links, or soft links, are files that act as pointers to other files or directories using a symbolic path reference. They are more flexible than hard links but can break if the target file is moved or deleted.

> ### Hard Links:

- Point directly *to the data (inode)* of the original file.
- Changes to the original changes all hard links.
- Cannot span different file systems.
- Can’t point to directories.
- **Syntax:** `ln target link(name)`
  - Example: `ln file1.txt ~/file2.txt`
  - This creates a hard link named `file2.txt` in **home directory** pointing to the same inode as `file1.txt`.

> ### Symbolic Links:

- Point *to the path of the target file*.
- Deleting or moving the original file breaks the symbolic link.
- Can span different file systems.
- **Syntax:** `ln -s target linkname`
  - Example: `ln -s /path/to/target /path/to/link`
  - This creates a symbolic link named `link` pointing to the file or directory `target`.

Example Scenario:

- Use hard links when you need multiple references to the same file, like organizing files or creating backups.

    - Use symbolic links when you want a flexible way to reference files across directories or file systems, such as linking configuration files or providing access to files from multiple locations.

Mounting:

1. **What is Mounting?**
    - Mounting is the process of connecting or attaching a storage device, like a USB drive or a network share, to your computer's existing filesystem. It's like making the contents of that device accessible to your computer's file system.
2. **Why Do We Need to Mount?**
    - Your computer's filesystem has a specific structure, like folders and directories. When you connect a new storage device, your computer needs to know where to put its files. By mounting the device, you tell your computer where to incorporate the files from that device into its existing file system structure.
3. **How Does It Work?**
    - When you mount a device, you're essentially telling your computer to treat the files on that device as part of its own files. You specify where these new files should appear in your existing file system structure.
4. **Example:**
    - Let's say you plug in a USB drive. Your computer doesn't automatically show its files because it doesn't know where to put them. By mounting the USB drive, you're telling your computer to make those files accessible. *It's like opening a drawer and placing files inside it.*
5. **Unmounting:**
    - When you're done with the storage device, you need to unmount it before physically disconnecting it. *This is like closing the drawer before taking it out*. It ensures that all the files are safely stored and that you don't lose any data.
        - This is equivalent to **ejecting** a storage device in Windows. Both processes ensure data integrity by completing all pending read/write operations before safe removal.

Quota:

Think of quotas as *limits* set on how much of a particular resource a user or group can consume on a system. In simple terms, it's like *putting a cap on how much of something someone can use*, done by the administrators.

1. **Resource Limits:**
    - Quotas are used to control and limit the amount of resources, such as disk space or file count, that a user or a group of users can consume on a system.
2. **Types of Quotas:**
    - **Disk Quotas:** These limit the amount of disk space a user or group can use on a filesystem.
    - **File Quotas:** These limit the number of files a user or group can create on a filesystem.
3. **Why Use Quotas?**
    - Quotas help in resource management and prevent users from monopolizing system resources.
    - They ensure fair usage of resources among multiple users, preventing one user from using up all available space.

> The `/etc/fstab` file (short for filesystem table) is a configuration file used by Linux operating systems to define how filesystems should be mounted and managed during system boot. While it is not directly related to disk quotas, it plays a crucial role in mounting filesystems with quota support enabled

⇒ **Example of `/etc/fstab` entry:**

```
/dev/sda1   /home   ext4   defaults,usrquota,grpquota   0   2
```

A breakdown of each component:

1. `/dev/sda1`:
    - Specifies the device or UUID representing the filesystem to be mounted.
2. `/home`:
    - Specifies the directory in the filesystem hierarchy where the filesystem should be mounted.
    - `/home` is the mount point.
3. `ext4`:
    - Specifies the type of filesystem present on the device.
    - Indicates that the filesystem is of the ext4 type.
4. `defaults,usrquota,grpquota`:

- Mount options for the filesystem, separated by commas.
- `defaults` : Includes default mount options provided by the system.
- `usrquota` : Enables user disk quotas on the filesystem.
- `grpquota` : Enables group disk quotas on the filesystem.

6. `0` :
    - Indicates whether the filesystem should be included in the `dump` backup utility.
    - A value of `0` means the filesystem will not be backed up by `dump` .
7. `2` :
    - Determines the order in which filesystems should be checked by the `fsck` filesystem consistency checker during system boot.
    - A value of `2` means the filesystem should be checked after filesystems with a `dump` flag of `1` .

▼ Lecture 8:

Processes:

- A program is a binary executable file, and a process is an instance of a running program.
- Multiple instances of processes from the same program CAN run simultaneously.
- In UNIX, each process must have a parent process, forming a hierarchical structure.
- Each process is identified by a unique Process ID (PID), typically a 5-digit number.

Process Management:

- **Competition:**
    - Or : *Multiple* users running *multiple* commands simultaneously on the *same* system.
    - Measures are necessary for CPU management and process switching to handle this competition efficiently, Done by the OS.
- Processes must continue to run even after the user who initiated them has logged out.

Types of Processes:

1. **Interactive Processes:**
    - Initiated and controlled through a terminal session where a user is connected and started them.

2. **Automatic (Batch) Processes:**
    - Not connected to a terminal.
    - Tasks can be queued into a spooler area for execution on a FIFO basis.
    - Scheduled to run at a certain date and time using the `at` command.
    - Scheduled to run when the total system load is low enough to accept extra jobs using the `batch` command.
3. **Daemons:**
    - Started when the system boots and executed continuously in the background.
    - Examples include:
        - **HTTPD** (web server daemon)
        - **SSHD** (Secure Shell daemon)
        - **MySQLD** (MySQL database daemon)
    - Daemons usually have names ending with 'd'.

Main Process States:

1. **Running (R):**
    - Indicates that the process is currently executing on the CPU.
2. **Sleeping (S):**
    - Indicates that the process is waiting for an event to complete. This event could be waiting for user input or for system resources to become available.
3. **Stopped/Traced (T):**
    - This state indicates that the process has been stopped, usually by receiving a signal. For example, during program debugging, a user may stop a process to inspect its current state.
4. **Zombie (Z):**
    - A terminated process that has finished execution but still has an entry in the process table, Also known as the situation where the child is there even though the parent process is killed. If a process remains in the zombie state for too long, it can consume system resources. Restarting the system may be necessary if the program needs to be run again. This state is considered abnormal.
    - Unkillable; because their parents are missing lol.
5. **Dead (X):**
    - Indicates a terminated process. This can occur if the process was killed or if it reached the last line of its source code.

N.B : If the parent process is in the Dead (X) state, all of its child processes are (must be) terminated as well.

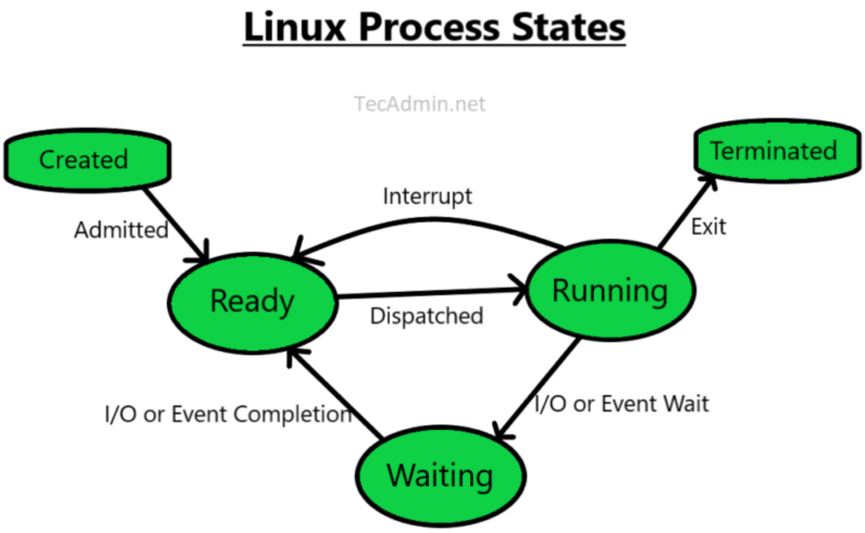
Example:

- When running `ps l` to display processes and their states, the STAT column indicates the current state of each process, Here is an example :

```
ashref@mint:~$ ps l
PID    UID    PPID   PGID    SID   TTY      STAT    TIME      COMMAND
1234   1000   567    1234   1234   tty1     S       00:05:00   bash
1      1000   0      567    567    tty1     S       00:10:00   init
890    1000   1234   890    1234   tty1     R       01:20:00   brave
```

init (systemd nowadays): short for initialization; the first process to start during system boot and has a process ID (PID) of 1.

Process States Diagram:



Explanation :

1. **Admitted (Creation):**

   - When a process is created, the operating system allocates necessary resources such as memory, assigns a unique Process ID (PID), and initializes process control blocks.

   - The process is added to the system's process table and enters the "Admitted" state.

- From here, the process moves to the "Ready" state when it's *ready to execute*.

2. **Dispatched (Ready to Running):**
    - In the "Ready" state, the process waits in the ready queue for the CPU to become available.
    - When the scheduler selects the process for execution, it transitions to the "Running" state.
    - The process's instructions are executed on the CPU, and it begins its task.
3. **Interrupted:**
    - During execution (Running), various events may occur that interrupt the process.
    - These events can include:
        - The expiration of the process's time slice.
        - Hardware interrupts.
        - The arrival of a higher-priority process.
        - ......
    - When an interrupt occurs, the process temporarily halts its execution and transitions to an appropriate state, such as "Blocked" or "Ready."
4. **(I/O or Event) Wait:**
    - While executing, a process may need to wait for input/output operations (I/O) to complete or for certain events to occur.
    - When waiting for I/O or events, the process transitions to the "Waiting(Blocked)" state.
    - The process is removed from the CPU's execution queue and waits for the I/O operation or event to finish.
5. **(I/O or Event) Completion:**
    - When the I/O operation or event the process is waiting for completes, the process transitions back to the "Ready" state.
    - If the process has the highest priority among the ready processes, it may be immediately dispatched onto the CPU.
    - Otherwise, it waits in the ready queue until the scheduler selects it for execution.
6. **Exit (Terminated):**

- When a process completes its task or explicitly terminates, it transitions to the "Exit" state.
- The operating system releases all resources associated with the process, including memory and system resources.
- The process's entry is removed from the system's process table, and it is no longer part of the system

Ending Processes:

- When a process ends normally, it *returns* an exit status to its parent process, indicating the outcome of its execution.
- This exit status is a *numerical value*, similar to the return value of the main function in programming.
- The practice of returning information upon job completion originates from the C programming language, upon which UNIX is built.
- Processes can also terminate due to receiving signals, which can be initiated using commands like `kill`.
    - Or by exiting; using `_exit()` system call, to free up the used resources.
- Signals offer a means of communication and control between processes and the operating system, allowing actions such as graceful termination or handling exceptional conditions.
- When a child process ends in Linux, its parent process acknowledges the termination using the `wait()` system call to retrieve the child's termination status.
- If the parent process dies before its child process:
    1. The child process becomes orphaned (Which is the normal situation, Zombie is abnormal).
    2. The orphaned child is adopted by the init process (process ID 1).
    3. Init becomes the new parent of the orphaned child.
    4. Init handles the cleanup of resources associated with the orphaned child.
    5. The orphaned child continues execution under the supervision of the init process.

Process Attributes:

- A process has several attributes, which can be viewed using the `ps` command:
    - **PID:** Process ID - Unique identifier assigned to each process.

    - **PPID:** Parent PID - ID of the parent process that spawned the current process.
    - **Nice Number:** Represents the degree of friendliness of the process towards other processes, influencing scheduling.
    - **Terminal / TTY:** Terminal to which the process is connected, if any.
    - **Username:** The user who owns the process.
    - **Primary Group:** The primary group of the user who started the process.

Terminals:

> N.B: the terminal isn't a process, so all the processes started within the terminal are supervised by thier parent process which is "The Shell" inside that terminal window.

Types of Terminals:

1. Regular Terminal Devices (TTY):
    - Native terminal devices for direct user interaction.
    - Examples include physical terminals, virtual consoles, and terminal emulators.
    - Users can input commands and receive output directly from the system.
2. Pseudoterminal Devices (PTS):
    - Virtual terminal devices emulating terminal behavior.
    - Commonly used for remote login sessions (SSH), terminal multiplexers (tmux, screen), and inter-process communication.
    - Provide a programmable terminal-like interface for processes.

> 💡 Processes are usually bound to terminals for interaction.

Means, if that process needs an input, or displays an output, this will be done in that terminal; Closing that terminal ⇒ Killing that process (Not all processes are bound to terminals tho).

▼ Printing a message from a terminal to another

1. Determine the terminal device of the target terminal where you want to print the phrase. You can find this information by running the ps command in the target terminal. For example:

```
ashref@kali:~$ ps
PTD  TTY       TIME     CMD
1234 pts/0     00:00:00 bash
1237 pts/1     00:00:00 bash
```

This will display the name of the terminal device, such as "/dev/tty1" or "/dev/pts/0".

2. Once you have the terminal device name, you can use the `echo` command to write the desired phrase to that terminal. For example, if the terminal device is "/dev/tty1", you can run:

```
ashref@kali:~$ echo "Your phrase here" > /dev/pts1
```

This will print your phrase in the other terminal.

⇒ The `/dev` directory contains special files representing hardware devices and pseudo-devices.

▼ Lecture 9:

`fork()`, `exec()`, and other related functions are *key system calls* in Unix-like operating systems, including Linux, used for process management. Here's a an explanation of each:

1. **fork():**

- Creates a new process (child) as a copy of the calling process (parent).
- Returns different values to parent and child processes.
- Child inherits attributes from the parent process.

▼ Usage

1. Call `fork()` to create a new child process.Check the return value:If it's negative, an error occurred.If it's zero, you're in the child process.If it's positive (i.e. the child's PID), you're in the parent process.

```
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t pid = fork();
    if (pid == -1) {
        // Error handling
    } else if (pid == 0) {
        // Child process
    } else {
        // Parent process
    }
    return 0;
}
```

2. **exec()**

- Replaces the current process image with a new one.
- Various variants like `execl()`, `execv()`, etc., with different parameter types.
- Used to run a new program in the context of the current process.

▼ **Usage**

1. After forking, call one of the `exec()` functions in the child process to replace the current process image with a new one.

```
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid == -1) {
        // Error handling
    } else if (pid == 0) {
        // Child process
            // Example: execute ls command
        execl("/bin/ls", "ls", "-l", NULL);
    } else {
        // Parent process
    }
    return 0;
}
```

3. **wait():**

- Parent process waits for child process termination.
- Retrieves termination status and exit code of child process.
- Allows for synchronization between parent and child processes.

▼ **Usage**

1. In the parent process, call `wait()` to wait for the child process to terminate.This allows the parent to synchronize with the child and retrieve its termination status.

```
#include <sys/wait.h>
#include <stdio.h>
```

```
int main() {
    pid_t pid = fork();
    if (pid == -1) {
        // Error handling
    } else if (pid == 0) {
        // Child process
    } else {
        // Parent process
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Child process terminated with exit status: %d\n", WEXITSTATUS(status));
        }
    }
    return 0;
}
```

4. **exit():**
    - Terminates the calling process.
    - Cleans up resources and exits with a specified exit status.
    - Typically called by processes when they complete execution.

    ▼ **Usage**

    1. Call `exit()` to terminate the current process.

    ```
    #include <stdlib.h>int main() {
        exit(0); // Terminate with exit status 0
    }
    ```

Process Scheduling:

1. **Automatic Processes:**
    - Start automatically without user intervention.
    - Examples: system services, daemons; those that the computer needs to do its work.
2. **Manual Processes:**
    - Started manually by users, typically through command-line or GUI.
    - Examples: running applications, executing scripts.

3.Scheduled Processes with `cron`:

- Time-based job scheduler.
- Automates tasks at predefined intervals or times.
- Configured using `cron` configuration files or user-specific crontab files.
- ***Two types of cron:***

  1. **crond:**
     - The cron daemon (`crond`) is a background process that runs continuously and is responsible for scheduling and executing cron jobs.
     - The cron daemon checks these files periodically (every minute) to determine when to execute scheduled tasks.
  2. **crontab:**
     - The `crontab` command is a utility used to create, modify, and manage cron jobs for individual users.
     - Users can use the `crontab` command to edit their personal crontab files, which contain their scheduled tasks.
     - Users can list existing cron jobs, add new ones, or remove existing ones using `crontab` commands.
     - Example (exam):

```
ashref@kali:~$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `cror
# command to install the new version when you edit this fi
# and files in /etc/cron.d. These files also have usernam
# that none of the other crontabs do.

SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OF
# |  |  |  |  |
# *  *  *  *  *  username  command to be executed
# Run the backup script every day at 2:00 AM
0 2 * * * root /path/to/backup.sh

# Run the cleanup script every Sunday at midnight
0 0 * * 0 root /path/to/cleanup.sh
```

> **To do so :**
>
> - Open the crontab editor:
>
>   ```
>   crontab -e
>   ```
>
> - Add the line (e.g.: to run the script every day at midnight):
>
>   ```
>   0 0 * * * /path/to/script.sh
>   ```
>
> - Save and exit the editor.

Linux Boot Sequence

Overview

- When the computer boots, the kernel is started.
- The kernel initiates the first process, typically the init process (pid = 1).
- The init process is responsible for managing all other processes.
- In Linux process management, there exists a parent-child relationship between processes.

The boot sequence in Linux involves several stages, starting from *“the BIOS POST”* and ending with *the “initialization of the init process”*, which is typically managed by systemd in modern distributions.

BIOS Overview

- **Definition**: The BIOS (Basic Input Output System) is firmware stored in ROM (Read-Only Memory) on the motherboard ⇒ cannot be modified.
- **Function**: It **initializes** the computer system after it is powered on, managing data flow between the OS and attached devices like HDDs, keyboards, printers…
- Uploaded by the manufacturer onto a chip on the motherboard.

- It's considered **firmware**, providing low-level control of computing device hardware.

1. BIOS POST (BIOS Power On Self Test)

During this initial stage of the boot process:

- The BIOS runs a POST (Power-On Self Test) to verify that all hardware components are functioning correctly.
- If the POST test fails, the computer may not be operable, and the boot process will halt.
- To access the BIOS settings, users typically press one of the function keys (e.g., F2, F10) depending on the machine during the boot process.

Summary of Modifiable Parameters in BIOS:

- **Boot Order**: Determines the sequence in which devices are checked for bootable operating systems.
- **System Time and Date**: Allows users to set the system clock.
- **CPU Settings**: Enables adjustments to CPU parameters such as clock speed and power management settings.
- **Device Configurations**: Provides options to configure attached devices like HDDs, SSDs, optical drives, etc.
- **BIOS Password**: A crucial parameter for security, it restricts access to BIOS settings and is stored in a small chip next to the BIOS on the motherboard.

> Although we stated earlier that the BIOS isn't modifiable, the idea of changing parameters in the BIOS might seem puzzling. This is because these parameters are stored in *a separate chip* near the CPU, powered by a small battery called the CMOS Battery.

2. Boot Loader

After a successful POST test, the boot process moves to the Boot Loader stage:

a. The BIOS loads and executes the boot code from the boot device, typically located in the first sector of the hard disk.

b. The boot loader presents the user with a boot screen, often offering multiple options to boot into various operating systems installed on the machine.

c. Once the user selects an option from the boot screen, the corresponding kernel is loaded into memory.

- **A well-known example** of a boot loader is GRUB2 (GRand Unified Bootloader Version 2).

3. Kernel Initialization

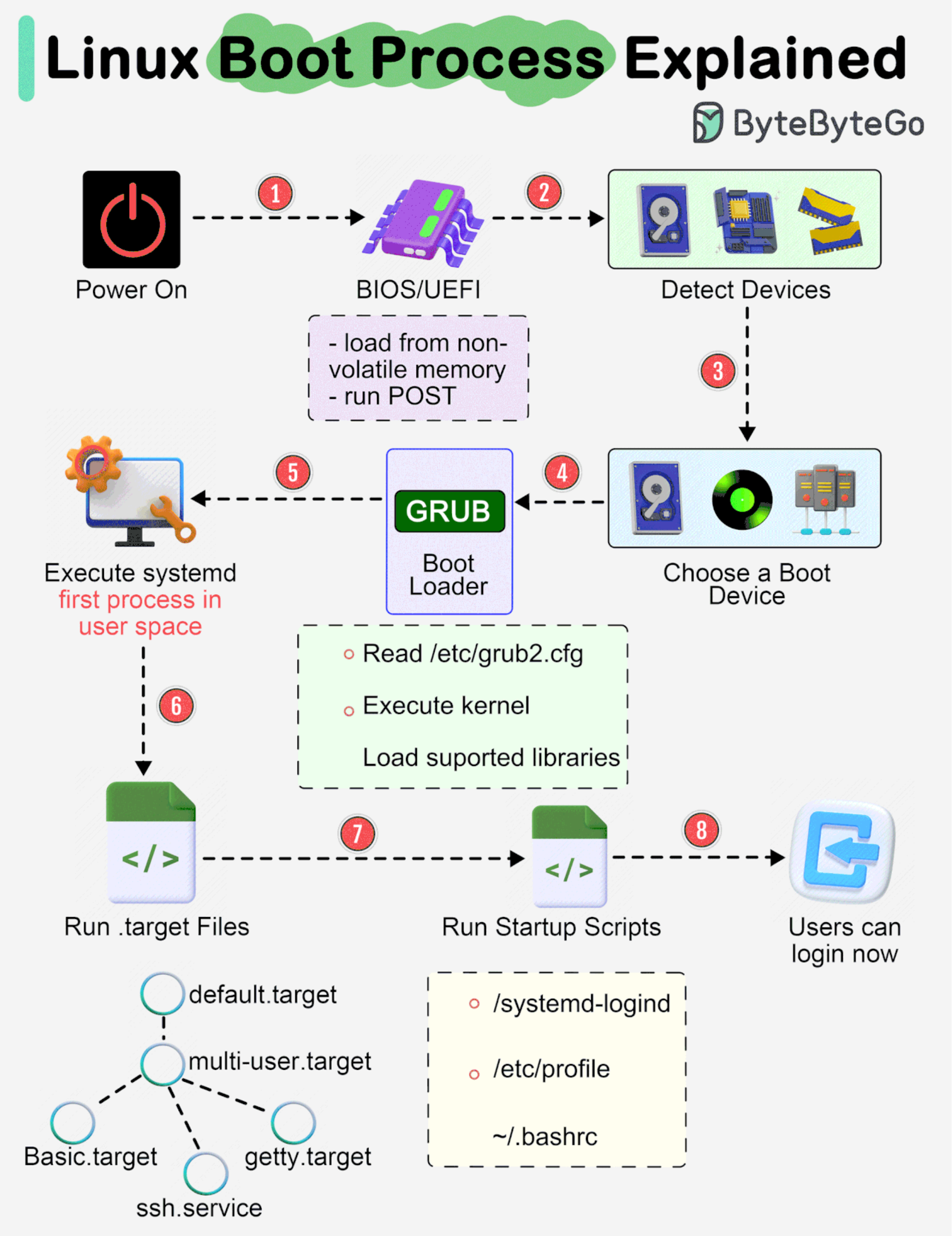
- After the boot loader loads the kernel into memory, the Kernel Initialization stage begins.
- The kernel is started, and it initializes various system components and drivers required for the functioning of the operating system.
- Hardware components such as the CPU, memory, storage devices, and input/output devices are identified and initialized during this stage.

4. init Process (systemd)

- In modern Linux distributions, the init process typically calls the systemd daemon.
- systemd is responsible for:
  - Mounting file systems
  - Starting and managing system services
- To check if systemd is the init process:

  ```
  ashref@kali:~$ ls -l /sbin/init
  ```

  - If systemd is used, there will be a pointer to `/lib/systemd/system` .



Additional Notes

- Important : Starting processes in Linux can be managed using commands such as `kill`, `nice`, `renice`, `top`, from ***lab4***.

Lecture 10:

Bash Shell Scripting

> A Bash script is a file containing a sequence of commands executed by the Bash program line by line. It allows for performing series of actions, such as navigating to a specific directory, creating folders, and launching processes using the command line. Unlike traditional programming languages, there's no need for compilation; instead, the Bash shell **interprets** the script directly.

Advantages:

- **Automation:** Automates system administration tasks, managing system resources, and performing routine operations, thereby saving time and reducing the risk of manual errors.
- **Portability:** Shell scripts can run on various platforms and operating systems, including Unix, Linux, MacOS, and even Windows through emulators or virtual machines.
- **Flexibility:** Highly customizable and easily modifiable to suit specific requirements.
- **Accessibility:** Can be edited using any text editor, and most operating systems have built-in shell interpreters.
- **Integration:** Can be integrated with other tools and applications, such as databases, web servers, and cloud services, enabling more complex automation.
- **Debugging:** Easy to debug with built-in debuggers available in most shells.

Script Naming Conventions:

- A Bash script is a text file with a `.sh` extension, *although* scripts can run fine *without it*.

Adding the Shebang:

```
#!/bin/bash
```

The Bash script starts with a shebang, a commented line (starting with `#`), indicating the path to the Bash interpreter.

Example Script:

```
#!/bin/bash

# Print the current date
echo "Current date is: $(date)"

# Prompt the user to enter the path of a directory
echo "Enter the path of a directory:"
read pathtodir

# Print files and folders in the specified directory
echo "Files and folders in $pathtodir are:"
ls -l "$pathtodir"
```

Ensure the script has executable permissions:

```
chmod u+x name.sh
```

To Execute the Script:

```
./name.sh
```

Alternatively, you can execute it using:

```
sh name.sh
```

or

```
bash name.sh
```

Comments:

In Bash scripting, comments are lines that begin with a `#` symbol. These lines are ignored by the interpreter and are solely for human readers.

- Comments are incredibly helpful for documenting code, explaining its functionality, and making it easier for others to understand.
- It's considered a best practice to include comments in your code, especially in complex scripts or when collaborating with others.

```
#A Comment, will be ignored by the interpreter
```

Note : Consider Adding them in the Exam !

Variables and Data Types in Bash

- Variables in Bash allow for storing and manipulating data throughout a script.
- Unlike some languages, Bash doesn't have strict data types; Variables can store numeric values, individual characters, or strings of characters.

Usage:

1. **Direct Assignment:**

   ```
   school="ENSIA"
   ```

2. **Assignment Based on Another Variable:**

   ```
   school_var= $school
   ```

⇒ N.B. : Comparing between Bash Shell Programming and General programming ain't that fair

Variable Access:

In Bash scripting, accessing variable values is essential. There are two common forms:

1. **$var**: Simply references the value of the variable `var`.
2. **${var}**: Allows for more controlled parsing:
   - Useful for concatenating variables with other strings without introducing white spaces.
     - Example: If `A="World"`, then `echo "Hello, ${A}!"` delimits the variable name `A` from `!`.
   - Used for parameter expansion, such as substring extraction or length calculation.
     - Example: `${A:0:3}` extracts the first three characters from string `A`.

> Better use the braces {}.

Variable naming conventions :

1. **Alphanumeric Characters**: Variable names can consist of letters, numbers, and underscores.
2. **Start with a Letter or Underscore**: Variable names must begin with a letter or an underscore.

3. **Case-Sensitive**: `var` and `VAR` would be treated as different variables.
4. **Avoid Special Characters**: such as spaces, punctuation marks, or arithmetic operators (`-`) in variable names.
5. **Avoid Bash Keywords**: e.g., `if`, `while`, `do` as variable names.
6. **Descriptive and Meaningful**: Choose descriptive and meaningful names that reflect the purpose of the variable.
7. **Uppercase Convention for Constants**: Conventionally, uppercase variable names are used for constants or variables with values that should not change during the script execution.
8. **Lowercase or CamelCase for Regular Variables**: Regular variables are typically named using lowercase letters or camelCase convention.

Quoting Mechanisms:

1. **Double Quotes ("")**:
   - Variables and special characters within double quotes are *interpreted* and *expanded* by Bash.
   - For example, `echo "Hello $USER"` would output `Hello` followed by the current username.

   ⇒ Double quotes allow for variable interpolation.
2. **Single Quotes ('')**:
   - Text within single quotes is treated literally, *without* any interpretation or expansion by Bash.
   - For example, `echo 'Hello $USER'` would output `Hello $USER` as is, without expanding `$USER`.

   ⇒ Single quotes prevent variable interpolation.

```
ashref@kali:~$ echo "Hello $USER"
Hello ashref

ashref@kali:~$ echo 'Hello $USER'
Hello $USER
```

N.B. : To force printing a special character, use the back slash “`\`”.

echo “It is \$10” ⇔ It is $10

Arithmetic Operations in Bash

Example Script:

```
#!/bin/bash

# Define variables
a=5
let b=3  #let  is similar to the (( )) , the variable will be an int by default

let c=(( 5+3 ))

# Perform arithmetic operations
let c = $a + $b          # Addition (via another variable)
difference=$((a - b))   # Subtraction (using arithmetic expansion)
product=$((a * b))      # Multiplication
quotient=$((a / b))     # Division
remainder=$((a % b))    # Modulus
increase=$(( a++ ))     # Or a=a+1
decrease=$(( a-- ))     # Or a=a-1


# Print the results
echo "Sum: $sum"
echo "Difference: $difference"
echo "Product: $product"
echo "Quotient: $quotient"
echo "Remainder: $remainder"
```

Explanation:

- `(( ))` is used to perform arithmetic operations in Bash.
- Inside `(( ))`, variables *do not need* to be prefixed with `$`.
- To assign to a variable, put the dollar sign before the parenthesis !
- The results are assigned to variables and can be printed using `echo`.

Run the Script:

1. Save the script to a file (e.g., `ao.sh`).
2. Make the script executable: `chmod u+x ao.sh`.
3. Run the script: `./ao.sh`.

Input and Output in Bash Scripts

Command Line Arguments:

In a Bash script or functions, command line arguments are accessed using positional parameters. Here's how it works:

- `$0`: Represents **the name of the script** or program being executed.
- `$1`, `$2`, ...: Represent the positional parameters, where `$1` is the first argument, `$2` is the second argument, and so on.

Other functions :

- `$*`: Represents **all** command line arguments as a single string, *separated by the first character of the* `IFS` *(Internal Field Separator) variable. By default, this character is a space.*
- `$@`: Represents all command line arguments as separate strings, preserving their original whitespace and quoting.
- `$#`: Represents **the number of command line arguments passed** to the script or function.
- `$$`: Represents the process ID (PID) of the current script or program.
- `$!`: Represents the process ID (PID) of the last background command executed.

Here's how you can access and use them in a script:

```
#!/bin/bash

# Accessing command line arguments
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments (as a single string): $*"
echo "All arguments (as separate strings): $@"
echo "Number of arguments: $#"
echo "Process ID of the current script: $$"
echo "Process ID of the last background command: $!"
```

⇒ In the above script, if you run `./script.sh 7 ENSIA`, it will output:

```
Script name: /script.sh
First argument: 7
Second argument: ENSIA
All arguments (as a single string): 7 ENSIA
All arguments (as separate strings): 7 ENSIA
```

```
Number of arguments: 2
Process ID of the current script: [Process ID]
Process ID of the last background command: [Empty or Previous Proce
```

Read Command:

Has many useful options :

- `-p prompt` : Displays the specified *prompt* before reading input.
- `-s` : Silent mode; input is not echoed to the terminal.
- `-n count` : Reads only `count` characters before stopping.
- `-r` : Prevents backslashes from being interpreted as escape characters.
- `-a array` : Reads input into an array variable, splitting words based on the value of the `IFS` (Internal Field Separator) variable.
- `-d delimiter` : Specifies a delimiter character to terminate input.
- `-t timeout` : Specifies a timeout in seconds. If no input is received within the specified time, the `read` command exits with a non-zero status.

Test Command:

- Also known as `[ ]`
- Always return either `True(0)` or `False(1)` . (Similar to the exit() context).

▼ Usage:

  - `test EXPRESSION`
  - `test EXPRESSION -a EXPRESSION` (logical AND)
  - `test EXPRESSION -o EXPRESSION` (logical OR)
  - `test !EXPRESSION` (logical NOT)

Example

```
#!/bin/bash

# Check if the current user is root
if test "$(whoami)" = "root"; then
    echo "You are the root user."
else
    echo "You are not the root user."
fi
```

Or Literally Similaly :

```
#!/bin/bash

# Check if the current user is root
if [ "$(whoami)" = "root" ]; then
    echo "You are the root user."
else
    echo "You are not the root user."
fi
```

Ternary-Like Condition

- `&&` : Executes the command on its right only if the command on its left succeeds.
- `||` : Executes the command on its right only if the command on its left fails.
    - Example : `test $A = “Hello” && echo true || echo false`

> using the `test` command or within `if` statements, you can check equality using either `=` or `==`. Both are correct.

test -testing integers:

- `-eq` : Tests if two integers are equal.
- `-ne` : Tests if two integers are not equal.
- `-lt` : Tests if the first integer is less than the second integer.
- `-le` : Tests if the first integer is less than or equal to the second integer.
- `-gt` : Tests if the first integer is greater than the second integer.
- `-ge` : Tests if the first integer is greater than or equal to the second integer.

```
#!/bin/bash

# Example: Testing if one integer is greater than another
num1=10
num2=5

if [ "$num1" -gt "$num2" ]; then
    echo "$num1 is greater than $num2"
elif [ $num -eq $num2 ]; then
        echo "Equal"
else
    echo "$num1 is not greater than $num2"
```

```
fi

# Example: Ternary-like behavior
num=10

# Ternary equivalent: echo "Number is greater than 5" if num > 5
# else echo "Number is not greater than 5"
[ $num -gt 5 ] && echo "Greater than 5" || echo "Smaller than 5"
```

test -testing file types:

- `-f`: Tests if a file exists and is a regular file.
- `-d`: Tests if a file exists and is a directory.
- `-e`: Tests if a file exists.
- `-s`: Tests if a file exists and is not empty.
- `-r`: Tests if a file exists and is readable.
- `-w`: Tests if a file exists and is writable.
- `-x`: Tests if a file exists and is executable.

```
#!/bin/bash

# Example: Testing file types
file="myfile.jpg"

if [ -f "$file" ]; then
    echo "$file exists and is a regular file."
fi

if [ -d "$file" ]; then
    echo "$file exists and is a directory."
fi
```

Switch Case:

```
#!/bin/bash

# Function to validate email address using regex
validate_email() {
 email=$1
 if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]
     echo "Valid email address: $email"
 else
     echo "Invalid email address: $email"
 fi
}

# Function to validate phone number using regex
validate_phone() {
    phone=$1
    if [[ $phone =~ ^\+213[0-9]{9}$ ]]; then
        echo "Valid phone number: $phone"
    else
        echo "Invalid phone number: $phone"
    fi
}

# Display menu options
echo "Welcome to the Data Validation Program:"
echo "1. Validate Email Address"
echo "2. Validate Phone Number"
echo "3. Exit"

# Prompt user for input
read -p "Enter your choice: " choice

# Process user input using case statement
case $choice in
    1)
        read -p "Enter email address: " email
        validate_email "$email"
        ;;
    2)
        read -p "Enter phone number (format: +213xxxxxxxxx): " phor
        validate_phone "$phone"
        ;;
    3)
        echo "Exiting the program. Goodbye!"
        exit 0
        ;;
    *)
        echo "Invalid choice. Please enter a number from 1 to 3."
        ;;
esac
```

Loops:

1. For Loop:

```
for item in [LIST]; do
    # Commands to execute for each iteration
done
```

Or Usual One :

```
#!/bin/bash

# Iterate over numbers from 1 to 5
for ((i = 1; i <= 5; i++)); do
    echo "Iteration $i"
done
```

Example:

```
#!/bin/bash

# Iterate over a list of numbers
for num in 1 2 3 4 5; do
    echo "Number: $num"
done
```

Example 2: (*)

```
#!/bin/bash

# Print a message indicating the start of the loop
echo "Looping through files and directories in the current director

# Iterate over each file (or directory) in the current directory
for f in *; do
# This line starts a loop where f iterates over each item (file or
# directory) in the current directory (* is a wildcard that matches
# all files and directories).
    if [ -f "$f" ]; then
        # a file?, print its name with a message that it's a file
        echo "Found file: $f"
    # Check if the current item is a directory
    elif [ -d "$f" ]; then
        # a dir?, print its name with a message that it's a directo
        echo "Found directory: $f"
    else
        # Neither a file nor a directory?, print a generic message
        echo "Found unknown item: $f"
    fi
done

echo "Loop finished."
```

Example 3: ($*)

```
#!/bin/bash

# Function to print each command-line argument separately
print_args() {
    echo "Printing each argument separately:"
    for n in $*; do
        echo "$n"
    done
}

# Call the function with command-line arguments
print_args arg1 arg2 "arg3 with spaces" arg4

# Output; Printing each argument separately:
# arg1
# arg2
# arg3
# with
# spaces
# arg4
```

N.B :

- `$n` : This syntax simply represents the value of the variable `n`. If `n` contains spaces or special characters, they will be treated as word separators by the shell.
- `"$n"` : When you enclose a variable in double quotes, like `"$n"`, it preserves the original whitespace and special characters in the value of the variable. It ensures

that the variable is treated as a single entity (i.e., a single argument), rather than being split into multiple words by the shell.

2. While Loop:

```
while [ CONDITION ]; do
    # Commands to execute while CONDITION is true
done
```

Example:

```
#!/bin/bash

# Print numbers from 1 to 5 using a while loop
num=1
while [ $num -le 5 ]; do
    echo "Number: $num"
    ((num++))
done
```

Example 2: (Reading from a file)

```
#!/bin/bash

# Define the file to read
file="example.txt"

# Check if the file exists
if [ ! -f "$file" ]; then
    echo "File $file does not exist."
    exit 1
fi

# Display a message indicating the start of reading
echo "Reading lines from $file:"

# Read each line from the file using a while loop
while IFS= read -r line; do
    echo "$line"
done < "$file"

# Display a message indicating the end of reading
echo "Finished reading lines from $file."
```

3. Until Loop:

```
until [ CONDITION ]; do
    # Commands to execute until CONDITION is true
done
```

Example:

```
#!/bin/bash

# Print numbers from 1 to 5 using an until loop
num=1
until [ $num -gt 5 ]; do
    echo "Number: $num"
    ((num++))
done
```

Continue and Linux:

```
#!/bin/bash

# Example using both break and continue
for ((i = 1; i <= 5; i++)); do
    if [ $i -eq 3 ]; then
        continue  # Skip iteration when i equals 3
    fi
    if [ $i -eq 4 ]; then
        break  # Exit the loop when i equals 4
    fi
    echo "Iteration $i"
done
echo "Loop finished"

#Output :
# Iteration 1
# Iteration 2
```

Functions:

Defining a Function:

```
function_name() {
    # Commands to be executed by the function
}
```

- A comment is required for each new function defined.

Example:

```
#!/bin/bash

# Define a function named greet
greet() {
    echo "Hello, world!"
}

# Call the greet function
greet
```

Function with Parameters:

```
function_name () {
    local parameter1="$1"  # Access the first parameter
    local parameter2="$2"  # Access the second parameter
    # Commands
}
```

▼ Variable Scope :

- In Bash all variable by default are defined as **global** variable, even if it was declared inside a function.
    - To avoid that; use the `local` keyword

Example:

```
#!/bin/bash

# Define a function named greet_with_name that accepts a paramet
er
function greet_with_name {
    local name="$1"
    echo "Hello, $name!"
}
```

```
# Call the greet_with_name function with an argument
greet_with_name "ENSIA"
```

Returning Values from Functions:

```
function_name() {
    # Commands
    return value
}
```

1. **Accessing the Return Value**:
    - After calling a function, you can access its return value using the special variable `$?`.
    - `$?` holds the exit status of the last executed command or function.
    - If the function executes successfully and returns a value, `$?` will hold that value.

Example:

```
#!/bin/bash

# Define a function named add that returns the sum of two numbers
add() {
    local num1="$1"
    local num2="$2"
    local sum=$((num1 + num2))
    return $sum
}

# Call the add function and store the result in a variable
add 5 3
result=$?  # After calling add(), $? holds the return value

echo "The sum is: $result"
```

Working With Files (From LAB5):

Task 1: Count Lines and Words in a File

```
#!/bin/bash

# Check if a filename is provided as an argument
if [ $# -ne 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

filename=$1

# Check if the file exists
if [ ! -f "$filename" ]; then
    echo "Error: File '$filename' not found."
    exit 1
fi

# Count the number of lines and words in the file
lines=$(wc -l < "$filename")
words=$(wc -w < "$filename")

# Print the counts
echo "Number of lines: $lines"
echo "Number of words: $words"
```

Task 2: Guess a Word

```
#!/bin/bash

# Loop until the word "end" is entered
while true; do
    # Prompt the user to enter a word
    read -p "Enter a word (type 'end' to finish): " word

    # Check if the word is "end"
        if [ "$word" == "end" ]; then  # Since word is a string;
        # its better to use "$word" isntead of $word, to keep an
y
        # whitespace or special characters within it
        echo "Exiting..."
        break
    fi
    # Otherwise, continue reading words
done
```

Task 3: Print Usernames

```
#!/bin/bash

# Print usernames of users on the machine
echo "Usernames on the machine:"
cut -d: -f1 /etc/passwd
```

Lecture 11:

Bash Shell Scripting (Part 2)

Advanced Conditional Expressions in Bash with `[[...]]`

Introduction to `[[...]]`

- The `test` command in Bash is commonly used for evaluating conditional expressions.
- `[[...]]` offers a more **robust** and **feature-rich** solution for writing conditional expressions.

Enhanced String Comparison

- With `[[...]]`, string comparison becomes more flexible.
- Use `==` and `!=` for pattern matching in string comparison.
  - Example:

    ```
    [[ $FILE == *.p ]] && cp "$FILE" scripts/
    ```

    - Explanation: If the file name matches the pattern ".*p*", copy it to the "scripts/" directory.

Logical AND and OR Operations

- Inside `[[...]]`, you can use `&&` and `||` for logical AND and OR operations.
- This allows for more concise and readable conditional expressions.

Regex Matching with `=~` Operator

- `[[...]]` supports regex matching using the `=~` operator.

- Some examples used for **Checking**:
- Example 1:

```
[[ $a =~ ^[0-9]*$ ]]  # If the variable $a contains 0 or more di
```

- Example 2:

```
[[ $a =~ ^[0-9]$ ]]  # If the variable $a is a single digit.
```

- Example 3:

```
[[ $a =~ ^[0-9]+$ ]]  # If the variable $a contains only digits.
```

More Examples:

1. **Check if a String Contains Only Digits:**

```
string="2024"
if [[ $string =~ ^[0-9]+$ ]]; then
    echo "The string contains only digits."
else
    echo "The string does not contain only digits."
fi
```

   - `^` denotes the start of the string, `[0-9]+` matches one or more digits, and `$` denotes the end of the string.

2. **Check if a String Starts with a Capital Letter:**

```
string="ENSIA"
if [[ $string =~ ^[A-Z] ]]; then
    echo "The string starts with a capital letter."
else
    echo "The string does not start with a capital lette
r."
fi
```

   - Explanation: This checks if the variable `string` starts with a capital letter (A-Z).

3. **Check if a String Contains a Specific Word:**

```
string="Hey from ENSIA Linux Lecture"
if [[ $string =~ \bLinux\b ]]; then
```

```
    echo "The string contains the word 'Linux'."
else
    echo "The string does not contain the word 'Linux'."
fi
```

- Explanation: This checks if the variable `string` contains the word "Linux".
- `\b` denotes a word boundary to ensure that "Linux" is a separate word and not part of another word.
  - For example, it would match "Linux" in `"Linux is here"` but would not match "Linux" in `"Linuxsomething”`

▼ Lecture 12:

AWK Command

- **Overview**:
  - A Powerful text-processing language and command in Unix systems.
  - First developed by Bell Labs in 1977.
  - Main authors: Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan.
  - Allows access to the AWK programming language, designed to process data within text streams.
  - ▼ AWK is actually a programming language.

```
awk 'BEGIN { print “Hello World!” }'
```

- Generally used to filter content from files.
  - **Syntax:** `awk` `'BEGIN { FS=" : " }` `{ print $0 }` `END { print NR }'` `/etc/passwd`
  - This code has 3 parts (up to 5 parts are possible):
    1. `BEGIN { FS = " : " }`: Begin Block; *optional*, executed only once, **before** any input line is processed. ⇒ Used to set up environment variables, initialize counters, or set the ***field separator.***
       - Specified either like `{ FS = ":" }` in the begin block, or as an *option* `-F":"`.
    2. `{ print $0 }`: Code Block; a series of commands to be executed for **each** line.
    3. `END { print NR }`: End Block; *optional*, executed only once, **after** all lines have been processed. ⇒ Used for any final calculations or cleanup (display summary data/results of processing).
    4. `/etc/passwd`: Input file.

- **Key Variables**:
    - `NR`: Number of lines that have been processed.
    - `$0`: The whole line.
    - `$1`: The first field.
        - `$2` : second field, and so on ...
- **Example**:

  ```
  awk 'END { print NR }' /etc/passwd
  ```

    - Prints the number of lines, similar to `wc -l /etc/passwd`.

Working on Selected Lines Only

- Only first 8 lines:

  ```
  awk 'NR < 9 { print NR, $1 }' /etc/passwd
  ```

- Process from line 5 to 11:

  ```
  awk 'NR >= 5 && NR <= 11 { print $0 }' /etc/passwd
  ```

- Process lines that end with the word "bash":

  ```
  awk '/bash$/ { print $0 }' /etc/passwd
  ```

- Print 1st, 3rd, and 7th fields (based on FS):

  ```
  awk 'BEGIN { FS = ":" } { print $1, $3, $7 }' /etc/passwd
  ```

- Improved print command using `printf` from **C-language**:

  ```
  awk 'BEGIN { FS = ":" } { printf "%10s %4d %17s\n", $1, $3,
  $7 }' /etc/passwd
  ```

- Process lines up to line 15 that do not start with '#':

  ```
  awk 'BEGIN { FS = ":" } !/^#/ && NR <= 15 { print $1, $3, $7
  }' /etc/passwd
  ```

- Print IDs greater than 400:

```
awk -F":" '$3 > 400 { print $3 }' /etc/passwd
```

Built-In Functions (Exactly those seen in C++ DSA1)

- **String Functions**: Handling and manipulating strings.
- **Numeric Functions**: Performing calculations and numeric operations.

User-Defined Functions

- Example:

  ```
  awk '
  function my_function(param1, param2) {
      # Function body
  }

  BEGIN {
      # Initialization code
      my_function(value1, value2)
  }
  '
  ```

- Save the AWK script to a file (`script.awk`):

  ```
  awk -f script.awk
  ```

▼ Commands:

1. **Navigation and Directory Management:**
   - `pwd` : Prints the current working directory.
   - `cd` : Changes the current directory.
   - `ls` : Lists directory contents.
   - `clear` : Clears the terminal screen.
   - `mkdir` : Creates a new directory.
     - `mkdir -m xyz name` **:** to specify the permissions.
       - `xyz` are the permissions in **Absolute Mode** (e.g. : 755)
   - `rmdir` : Removes a directory (if empty).
   - `df` : Displays disk space usage of filesystems.

- `ln` : create a hard link.
    - `ln -s` : create a symbolic link.
- `diff` : compare between the contents of two files line by line.

2. **Files and Directories Operations:**

- `touch` : Creates an empty file or updates the access and modification times of a file.
- `cp` : Copies files or directories.
- `mv` : Moves or renames files or directories.
- `rm` : Removes (deletes) files or directories.
- `head` : Outputs the first part of files. (10 if not specified { `-n x` }).
- `tail` : Outputs the last part of files. (10 if not specified { `-n x` }).
- `chmod` : Changes file permissions.
- `chown` : Changes file owner and group.
- `chgrp` : Changes group ownership of files.
- `sfdisk` : To display the partition table of a disk:
- `debugfs` : To interactively debug an ext4 filesystem:
- `badblocks` : To scan a device for bad blocks:
- `dosfsck` : To check and repair a FAT filesystem:
- `mkdosfs` : To create a FAT filesystem on a partition:
- `fdisk` : To partition a disk interactively.
    - `fdisk -l` : display current disk structure.
- `fsck` : To check and repair a filesystem:
- `mkfs` : To create an ext4 filesystem on a partition:
- `parted` : To create a new partition on a disk:
- `mount` : To mount a filesystem. (attach the filesystem found on a device to the system's file hierarchy)
    - Example: `mount /dev/sdb1 /mnt`
        - This command mounts the filesystem from the device `/dev/sdb1` to the directory `/mnt`
    - `mount [options] <device> <mount_point>` :
        - The `<device>` argument can refer to various types of storage devices.

        - The `<mount_point>` argument specifies the directory in the existing filesystem where the contents of the device will be made available.
        - ***Options:***
            - `-t <type>`: Specifies the filesystem type. If not specified, the type is determined automatically.
            - `-o <options>`: Allows specifying mount options like read-only (`ro`), read-write (`rw`), or specific permissions.
            - `-n`: Mounts the filesystem without updating `/etc/mtab` or `/proc/mounts`. Useful for temporary mounts.
            - `-r`: Mounts the filesystem read-only.
            - `-o remount`: Remounts a mounted filesystem with different options.
            - `-o bind`: Remounts a subtree somewhere else without moving the subtree's data.
- `unmount`: used to detach a mounted filesystem from the filesystem hierarchy, allowing you to safely remove storage devices or unmount network shares.
    - `sudo unmount /mountingpoint`
- `df -h` : Displays information about mounted filesystems and their disk usage
- `quota`: View quota limits and usage for the current user.
    - `quotacheck`: Checks and updates disk usage and quota information for filesystems.
        - `sudo quotacheck -avug /path(of file system)`
    - `sudo edquota -u username` :(Set user quotas interactively)
    - `sudo edquota -g groupname`: (Set group quotas interactively)
    - Edit `/etc/quotatab` to define default quotas for new users/groups.
    - `quota -u username` (View quota information for a specific user)
    - `quota -g groupname` (View quota information for a specific group)
    - `repquota /mountpoint(path(of filesystem))` (Report quota usage and limits for all users on a file system)
    - `sudo quotaon /path`: Enables quotas on specified file systems.
    - `sudo quotaoff /path`: Disables quotas on specified file systems.
    - `quotacheck`: Checks and repairs quota inconsistencies on filesystems.

3. **User and Group Management:**

- `id` : Displays the user and group IDs of the current user.
- `usermod` : Modifies user account attributes.
- `finger` : Displays user information, including login name, real name, shell, etc.
- `chfn` : Changes the user's full name information.
- `chsh` : Changes the user's login shell.
- `useradd` : Creates a new user account.
- `groupadd` : Creates a new group.
- `whoami` : Prints the current user's username.
- `userdel` : Deletes a user account.
- `adduser` : *Interactive* command to add a user.
- `deluser` : Deletes a user account.

4. **System Information:**
    - `hostname` : Prints or sets the system's hostname.
    - `date` : Prints or sets the system date and time.
    - `cal` : Displays a calendar.
    - `who` : Displays information about users who are currently logged in.
    - `last` : Shows listing of last logged in users.
    - `lastb` : Shows listing of last logged in bad users.
    - `users` : Lists users currently logged in.
    - `lslogins` : Displays information about known users in the system.
    - `whereis` : possible locations of file.
    - `route` : display infos about network of the machine.
    - `uptime` : time since the machine is up.
    - `w` : which users are online.
5. **Processes:**
    - `ps` : Displays information about active processes.
        - `ps aux` : Display information about all processes
        - `ps -u user` : Display process information for a specific user:
        - `ps axjf` : Display process information in a tree format
    - `top` : Provides dynamic real-time information about running processes and system resource usage.

    - `top` **then** `Shift + P` : Display processes sorted by CPU usage (similarly, can filter any output)

5. **Package Management:**
    - `synaptic` : A graphical package management tool used in some Linux distributions to manage software packages.
    - `apt` : Advanced Package Tool, used for package management on Debian-based systems.
6. **Text Editing:**
    - `vi` : A text editor with powerful editing capabilities.
    - `nano` : A simple text editor commonly used in terminal environments.
7. **File Compression/Manip:**
    - `gzip` : Compresses files using the gzip compression algorithm.
    - `gunzip` : Decompresses gzip-compressed files.
    - `zcat` : Decompresses and displays compressed files.
    - `tar` : Manipulates archives in the tar format.
        - `tar cf *.tar` : create a tar file.
        - `tar -xvf filename` : extract content from a file.
    - `wget` : Downloads files from the internet
8. **Command Line Utilities:**
    - `echo` : Prints text to the terminal.
    - `;` : Separates multiple commands on a single line.
    - `&&` : Executes the next command only if the previous command succeeds.
    - `>` : Redirects command output to a file, overwriting the file's contents.
    - `>>` : Appends output to a file.
    - `<` : Redirects input from a file to a command.
    - `wc` : Counts lines, words, and characters in a file or input stream.
    - `grep` : Searches for patterns in files or input streams.
    - `less` : A paginator for viewing text files, allowing scrolling and searching.
    - `cat` : Concatenates and displays files' content.
    - `apropos` : Searches the manual page names and descriptions for a specified keyword.
    - `unalias` : Removes aliases previously set with the 'alias' command.

- `sort` : Sorts lines of text files.
    - `sort -u` : stands for *unique*; ensures only unique lines are retained in the output.
- `cut` : Extracts sections from each line of files.
- `type` : Indicates how a command name is interpreted, and its type.
- `whatis` : Displays one-line manual page descriptions.
- `info` : Provides detailed information about commands and concepts.
- `tee` : to read an input.

9. **Environment and Shell:**
    - `export` : Sets environment variables for child processes.
    - `alias` : Creates shortcuts for commands or command sequences.
    - `help` : Provides help for shell built-in commands.
    - `exit` : Terminates the current shell session.
    - `$PATH` : Directories where shell searches for executable files.
        - to modify : `export PATH=…`

10. **Regular Expressions:**

> Or (regex); Powerful tools for pattern ***matching*** and manipulation of text data. In Linux, regular expressions are widely used in various contexts such as searching, replacing, and filtering text within files or as part of command-line utilities like `grep` , `sed` , `awk` , and also witihn Bash Itself using `[[ ]]` conditional expressions.

- **Metacharacters**: a character that has special meaning.
    - `.` : Matches any **single** character (except newline).
    - `*` : Matches **0 or more** occurrences of the preceding character or group.
    - `+` : Matches **1 or more** occurrences of the preceding character or group.
    - `?` : Matches **one** occurrence of the preceding character or group.
    - `^` : Anchors the regex to the **start** of the line.
    - `$` : Anchors the regex to the **end** of the line.
    - `[ ]` : Matches any single character within the brackets.

        - `[abc]` matches any single character that is either 'a', 'b', or 'c'.
            - `[^abc]` : Complementary of previous; Matches any character excpet 'a' , 'b' or 'c'. (Different than `^[abc]` !)
        - `[0-9]` matches any single digit from 0 to 9.
        - `[a-zA-Z]` matches any single alphabet character, either lowercase or uppercase.
    - `{ }`: Specifies the number of occurrences of the preceding character or group.
    - `|`: Alternation, matches either the expression before or after it.
    - `\`: Escapes a metacharacter to be treated as a literal character.
- **Groups and Ranges** :
    - (abc) : Matches the exact sequence "abc"; treats multiple characters as **a single unit**.
    - a{2,4} : Matches "a" repeated 2 to 4 times.
        - Example : **Using `grep` to Find IP Addresses**

    ```
    grep -Eo '\b([0-9]{1,3}\.){3}[0-9]{1,3}\b' log.txt
    ```

- *Example:*
    - `ls ???[lnx]*` : print all the words with 4th char being l, n or x.
    - the pattern `“^ens.a$”` matches all the words starting with **ens** ending with **a** and 4th character being any single character.
    - `grep -E ‘^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+` `\` `.[a-zA-Z]{2,}$’ file.txt` : search for valid email adresses.

11. **Sed Command:**

    `sed`, short for stream editor, is a powerful command-line tool used for text **processing** and manipulation. It reads text from a file or standard input, performs operations (such as search, replace, insert, delete), and prints the results to standard output. `sed` operates by applying commands to each line of input or to a range of lines specified by addresses

    Usage:

    `sed [options] [commands] [input-file...]`

    Examples:

    1. **Substitution:**

```
sed 's/old_text/new_text/g' filename.txt
```

This command **replaces all** occurrences of "old_text" with "new_text" in the file "filename.txt".

2. **Delete lines:**

   ```
   sed '/pattern/d' filename.txt
   ```

   This command **deletes** all lines containing the specified pattern from the file "filename.txt".

3. **Insert text before a line:**

   ```
   sed '/pattern/i\new_line' filename.txt
   ```

   This command inserts "new_line" before each line containing the specified pattern in "filename.txt".

4. **Displaying Specific Lines:**

   ```
   sed -n '5,10p' filename.txt
   ```

5. **Appending Text After Specific Lines:**

   ```
   sed '5a\new_line' filename.txt
   ```

6. **Specifying Line Ranges:**

   ```
   sed '10,20d' filename.txt
   ```

   This command deletes lines 10 through 20 from the file "filename.txt"

7. **Print Last Line:**

   ```
   sed -n '$p' filename.txt
   ```

8. **Mixing it with regular Expressions:**

   a. **Print lines of f1.txt that start with "Ash":**

      ```
      sed -n '/^Ash/p' f1.txt
      ```

   b. **Print lines of f1.txt that end with "AI":**

```
sed -n '/AI$/p' f1.txt
```

c. **Add the string "Line: " at the beginning of each line in f1.txt:**

```
sed 's/^/Line: /' f1.txt
```

d. **Put the word "courage" between double quotes:**

```
sed 's/courage/"&"/g' f1.txt
```

e. **Add a comma at the end of each line:**

```
sed 's/$/,/' filename.txt
```

f. **Convert tabs to spaces:**

```
sed 's/\t/ /g' filename.txt
```

g. **Delete empty lines:**

```
sed '/^$/d' filename.txt
```

This command deletes lines that are empty (contain only whitespace characters).

Options:

- `-e <script>`: Add the script to the commands to be executed.
- `-n`: Suppress automatic printing of pattern space.
- `-i`: Edit files in place.
- `-r` or `-E`: Use extended regular expressions.
- `-f <script-file>`: Add the contents of the script-file to the commands to be executed.

> And with that, The Introduction to Linux course comes to an end. Keep on coding, keep on exploring, And rebi ywefe9 ✨.
>
> **Berbaoui Ashref**