Unit 2 : Basic Linux Commands

2.1 Directory Navigation Commands

The Linux file system is organized in a hierarchical (tree-like) structure, starting from the root directory (/). Understanding how to move around and manage directories is crucial.

1. pwd (Print/Present/Current Working Directory)

Purpose: The pwd command is used to display the full, absolute path of the current working directory (where you are currently located in the file system).

Syntax:

```
$ pwd
```

Output Example:

```
/home/user/documents
```

Key Point: This command is very useful when you get "lost" in the file system and need to quickly know your current location.

2. cd (Change Directory)

- **Purpose:** The cd command is used to change your current working directory. It allows you to move to different locations within the file system.

Syntax:

```
cd [directory_path]
```

Key Concepts:

- **Absolute Path:** A path that starts from the root directory (/). It provides the complete address of a directory from the top of the file system.
  - **Example:** cd /home/user/projects
- **Relative Path:** A path that is relative to your current working directory. It does not start with /.

Common cd Usages:

- cd ~ or cd: Go to your **home directory**. This is the default directory for your user when you log in.
- cd .. : Go up **one level (One level Back)** (to the parent directory).
- cd . : Stay in the **current directory**. (Not very useful on its own, but conceptually important for relative paths).

- cd - : Go to the **previous directory** you were in. Very handy for quickly switching between two directories.
- cd / : Go to the **root directory**.

Examples:

```
pwd                 # Output: /home/youruser
cd documents        # Change to 'documents' inside your home directory
pwd                 # Output: /home/youruser/documents
cd ..               # Go up one level
pwd                 # Output: /home/youruser
cd /etc             # Go to the absolute path /etc
pwd                 # Output: /etc
cd ~                # Go back to your home directory
pwd                 # Output: /home/youruser
```

3. mkdir (Make Directory)

Purpose: The mkdir command is used to create new directories (folders).

Syntax:

```
$ mkdir [options] directory_name(s)
```

Common Options:

- -p or --parents: Creates parent directories if they don't already exist. This is very useful for creating nested directories in one go.

Examples:

```
mkdir my_new_folder           # Creates 'my_new_folder' in the current directory
mkdir folder1 folder2 folder3 # Creates multiple directories at once
mkdir -p project/src/main/java  # Creates 'project', then 'src' inside 'project', etc.
                              # If 'project' doesn't exist, it will be created.
```

- **Note:** Directory names can contain spaces, but then they must be enclosed in quotes (e.g., mkdir "my documents"). It's generally good practice to avoid spaces in directory and file names in Linux.

4. rmdir (Remove Directory)

Purpose: The rmdir command is used to delete **empty** directories.

Syntax:

rmdir [options] directory_name(s)

- **Important Note:** rmdir will *fail* if the directory is not empty. If you need to remove non-empty directories, you'll typically use rm -r (which will be covered in File Management Commands).

Examples:

```
$ mkdir empty_folder          # Create an empty folder
$ rmdir empty_folder          # Successfully remove it


$ mkdir non_empty_folder
$ touch non_empty_folder/file.txt # Create a file inside it
$ rmdir non_empty_folder      # This will fail with an error like:
```

5. ls (List Directory Contents)

Purpose: The ls command is used to list the contents of a directory. It shows files and subdirectories within the specified (or current) directory.

Syntax:

ls [options] [directory_path]

- **Common Options:**
  - -l: **Long listing format**. Shows detailed information (permissions, number of links, owner, group, size, modification date, name).
  - -a: **All files**. Shows hidden files (files starting with a .).
  - -h: **Human-readable sizes** (with -l). Displays file sizes in K, M, G, etc.
  - -r: **Reverse order**. Sorts in reverse order.
  - -t: **Sort by modification time**, newest first.
  - -R: **Recursive listing**. Lists contents of subdirectories as well.
  - -F: Appends a character to entries to indicate their type (/ for directories, * for executables, @ for symbolic links, etc.).

Examples:

ls # Lists contents of the current directory

ls -l # Long listing of current directory

ls -a # Shows all files, including hidden ones

ls -lh # Long listing with human-readable sizes

ls -ltr # Long listing, sorted by time (newest last), reverse order

ls /etc # Lists contents of the /etc directory

ls -R my_project_dir # Lists contents of 'my_project_dir' and all its subdirectories

Output Interpretation (for ls -l):

-rw-r--r-- 1 user group 1024 Jul 7 10:30 myfile.txt

drwxr-xr-x 2 user group 4096 Jul 6 15:00 my_folder/

- First character (- or d): File type (- for regular file, d for directory, l for symbolic link, etc.)
- Next 9 characters (rw-r--r--): File permissions (user, group, others)
- 1: Number of hard links
- user: Owner username
- group: Owner group
- 1024: File size in bytes
- Jul 7 10:30: Last modification date and time
- myfile.txt: File or directory name

6. tree

Purpose: The tree command lists the contents of directories in a tree-like format, showing the hierarchical structure.

Syntax:

tree [options] [directory_path]

- **Note:** tree is often not installed by default on all Linux distributions. You might need to install it using your distribution's package manager (e.g., sudo apt install tree on Debian/Ubuntu, sudo yum install tree on RedHat/CentOS, sudo dnf install tree on Fedora).

- **Common Options:**
  - -L level: Descends only level directories deep.
  - -F: Appends a character to entries to indicate their type (similar to ls -F).
  - -d: List directories only, not files.
  - -a: All files (including hidden ones).

Examples:

tree # Displays the tree structure of the current directory

tree -L 2 # Displays 2 levels deep from the current directory

tree /home/user/documents # Displays the tree structure of a specific directory

tree -d # Shows only directories in the tree structure

Output Example:

```
├── project
│   ├── src
│   │   ├── main
│   │   │   └── App.java
│   │   └── test
│   │       └── AppTest.java
│   └── pom.xml
├── documents
│   ├── report.pdf
│   └── notes.txt
└── backup
    └── old_files/
4 directories, 5 files
```

2.2 File Management Commands

These commands are fundamental for manipulating files within the Linux file system.

1. cat (Concatenate and Display Files)

- **Purpose:** The cat command has multiple uses, primarily to:
  - Display the content of one or more text files to the standard output (your terminal).
  - Concatenate (combine) multiple files into a single output or a new file.
  - Create a new file (though touch is more common for creating empty files, and text editors for content) using '>' sign.
  - We can append the file…Using '>>' Sign.
- **Syntax:**

  cat [options] [file(s)]
- **Common Options:**
  - -n or --number: Number all output lines.
  - -b or --number-nonblank: Number non-blank output lines.
  - -s or --squeeze-blank: Suppress repeated empty output lines.
- **Examples:**

cat myfile.txt # Displays the content of myfile.txt

cat file1.txt file2.txt # Displays content of file1.txt then file2.txt

cat -n mylog.txt # Displays mylog.txt with line numbers

cat file1.txt file2.txt > combined.txt # Concatenates file1.txt and file2.txt into a new file called combined.txt.

(The '>' symbol redirects output to a file)

cat > new_file.txt # Creates a new file. Type content, then press Ctrl+D to save and exit.

2. rm (Remove)

- **Purpose:** The rm command is used to delete files and directories. It's a powerful command, and caution should be exercised as deleted files are generally not

recoverable from the command line (unless you have a backup or a specific recovery tool).

- **Syntax:**

rm [options] file(s) or directory(s)

- **Common Options:**
  - -i or --interactive: **Interactive mode**. Prompts you before every removal. Highly recommended for safety, especially when deleting multiple files or directories.
  - -f or --force: **Force removal**. Ignores non-existent files and never prompts. **Use with extreme caution!** This can be dangerous if used incorrectly.
  - -r or -R or --recursive: **Recursive removal**. Deletes directories and their contents (subdirectories and files). This is necessary for deleting non-empty directories..
  - -d :removes only empty directory
- **Examples:**

rm myfile.txt # Deletes myfile.txt

rm -i another_file.txt # Prompts before deleting another_file.txt

rm -f unwanted_file.txt # Deletes unwanted_file.txt without prompting

rm -r my_empty_dir # Deletes an empty directory (rmdir is safer for empty ones)

rm -r my_project_folder # Deletes my_project_folder and all its contents (files and subdirectories)

rm -rvf my_old_project # Forcefully and recursively deletes my_old_project, showing verbose output

- **Warning:** Be very careful with rm -rf / or rm -rf * (especially as root or in critical directories), as these can delete your entire system or important data without warning. Always double-check your command before pressing Enter.

3. cp (Copy)

- **Purpose:** The cp command is used to copy files and directories from one location to another.
- **Syntax:**

cp [options] source_file destination_file

cp [options] source_file(s) destination_directory

cp [options] source_directory destination_directory

- **Key Concepts:**
  - **Source:** The file or directory you want to copy.
  - **Destination:** Where you want the copy to go.
    - If the destination is a file name, the source file is copied and renamed to the destination file name.
    - If the destination is an existing directory, the source file(s) are copied into that directory with their original names.
- **Common Options:**
  - -i or --interactive: **Interactive mode**. Prompts before overwriting an existing file.
  - -r or -R or --recursive: **Recursive copy**. Required when copying directories. It copies the directory and all its contents.
  - -v or --verbose: Explains what is being done.
  - -u or --update: Copy only when the SOURCE file is newer than the destination file or when the destination file is missing.
  - -a or --archive: **Archive mode**. Preserves file attributes (permissions, ownership, timestamps, symbolic links, etc.). This is often preferred for backups. It's equivalent to -dR --preserve=all.
- **Examples:**

cp myfile.txt backup/ # Copies myfile.txt into the 'backup' directory

cp myfile.txt new_name.txt # Copies myfile.txt and renames the copy to new_name.txt

cp file1.txt file2.txt docs/ # Copies file1.txt and file2.txt into the 'docs' directory

cp -i report.txt archive/ # Copies report.txt, prompts if 'report.txt' already exists in 'archive'

cp -r my_project_dir backup/ # Copies the entire 'my_project_dir' and its contents to 'backup'

cp -a website/ public_html/ # Copies 'website' and its contents to 'public_html', preserving attributes.

4. mv (Move)

- **Purpose:** The mv command is used to move files and directories from one location to another, or to rename them. Unlike cp, mv does not create a new copy; it moves the original.
- **Syntax:**

  mv [options] source_file destination_file

  mv [options] source_file(s) destination_directory

  mv [options] source_directory destination_directory
- **Key Concepts:**
  - **Renaming:** If the source and destination are in the *same* directory, mv acts as a rename command.
  - **Moving:** If the source and destination are in *different* directories, mv moves the file/directory.
- **Common Options:**
  - -i or --interactive: **Interactive mode**. Prompts before overwriting an existing file.
  - -f or --force: **Force move**. Overwrites existing files without prompting.
  - -v or --verbose: Explains what is being done.
  - -u or --update: Move only when the SOURCE file is newer than the destination file or when the destination file is missing.
- **Examples:**

mv old_name.txt new_name.txt # Renames old_name.txt to new_name.txt in the same directory

mv report.pdf documents/ # Moves report.pdf into the 'documents' directory

mv -i file_to_move.txt existing_file.txt # Prompts before overwriting existing_file.txt

mv -v my_dir /tmp/ # Moves 'my_dir' to '/tmp/', showing verbose output

mv file1.txt file2.txt file3.txt archive/ # Moves multiple files to 'archive'

5. touch

- **Purpose:** The touch command is primarily used for two main purposes:
  - To **create new, empty files**.
  - To **update the access and modification timestamps** of existing files or directories without changing their content.
- **Syntax:**

  touch [options] file(s)
- **Common Options:**
  - -a: Change only the access time.
  - -m: Change only the modification time.
  - -c or --no-create: Do not create any files.
  - -r reference_file: Use the access/modification times of reference_file instead of the current time.
  - -t [[CC]YY]MMDDhhmm[.ss]: Use a specified timestamp instead of the current time.
- **Examples:**

touch new_empty_file.txt # Creates a new empty file named new_empty_file.txt

touch file1.txt file2.txt # Creates two new empty files

touch existing_file.txt # Updates the access and modification times of existing_file.txt to the current time

ls -l existing_file.txt # Check timestamps before and after using touch

touch -r template.txt draft.txt # Sets the timestamps of draft.txt to match template.txt

touch -t 202407081030.00 old_document.txt # Sets timestamp of old_document.txt t

2.3 File Permissions and Ownership

Linux is a multi-user operating system, and a robust permission system is essential to control who can access, modify, and execute files and directories.

Understanding Linux File Permissions

Every file and directory in Linux has associated permissions that dictate what actions different types of users can perform. These permissions are typically shown when you use ls -l.

Example ls -l output:

-rw-r--r-- 1 user group 1024 Jul 7 10:30 myfile.txt

drwxr-xr-x 2 user group 4096 Jul 6 15:00 my_folder/

The first block of 10 characters represents the file type and permissions:

1. **First Character (File Type):**
    - -: Regular file
    - d: Directory
    - l: Symbolic link
    - (There are others like c for character devices, b for block devices, etc., but these are less common for general users).
2. **Next Nine Characters (Permissions):** These are divided into three sets of three characters each:
    - **User (Owner) Permissions:** The first three characters.
    - **Group Permissions:** The middle three characters.
    - **Others (World) Permissions:** The last three characters.

Each set has three possible permissions, represented by r, w, x, or - if the permission is absent:

- r: **Read** permission
- w: **Write** permission
- x: **Execute** permission

Meaning of Permissions:

- **For Files:**
  - r (Read): Allows viewing the content of the file.
  - w (Write): Allows modifying (editing, saving, deleting) the file.
  - x (Execute): Allows running the file as a program (if it's an executable script or binary).
- **For Directories:**
  - r (Read): Allows listing the contents of the directory (ls).
  - w (Write): Allows creating, deleting, or renaming files *within* the directory.
  - x (Execute): Allows entering/traversing the directory (cd) and accessing its files and subdirectories. Without 'x' on a directory, you cannot cd into it or access anything inside, even if you have read permission.

Numeric (Octal) Representation of Permissions:

Each permission (r, w, x) also has a numeric value:

- r = 4
- w = 2
- x = 1
- - = 0

To get the numeric representation for a set of three permissions, you sum their values:

- rwx = 4 + 2 + 1 = 7 (Read, Write, Execute)
- rw- = 4 + 2 + 0 = 6 (Read, Write)
- r-x = 4 + 0 + 1 = 5 (Read, Execute)
- r-- = 4 + 0 + 0 = 4 (Read Only)
- --- = 0 + 0 + 0 = 0 (No Permissions)

So, permissions like rwxr-xr-x can be represented numerically as 755:

- User: rwx = 7
- Group: r-x = 5
- Others: r-x = 5

1. chmod (Change Mode)

- **Purpose:** The chmod command is used to change file and directory permissions.
- **Syntax:**

  chmod [options] mode file(s) or directory(s)

- **Modes:** Permissions can be set using two main methods:
  1. **Symbolic Mode (UGOA):** Uses characters to specify who (u, g, o, a), what operation (+, -, =), and what permission (r, w, x).
     - u: user (owner)
     - g: group
     - o: others
     - a: all (u + g + o)
     - +: add permission
     - -: remove permission
     - =: set permission exactly (override existing)
  2. **Numeric (Octal) Mode:** Uses the 3-digit (or 4-digit) octal representation of permissions. This is generally preferred for clarity and consistency.
- **Common Options:**
  - -R or --recursive: Change permissions of files and subdirectories recursively.
  - -v or --verbose: Show detailed information for every changed file.
- **Examples (Symbolic Mode):**

chmod u+x myfile.sh # Add execute permission for the owner

chmod g-w myfile.txt # Remove write permission for the group

chmod o=r myfile.txt # Set read permission for others, remove all others

chmod a+rw my_file.txt # Add read and write for all (user, group, others)

chmod ug+w,o-rwx my_dir # Add write for user/group, remove all for others on 'my_dir'

chmod +x myscript.sh # Adds execute permission for everyone if no target (u,g,o) is specified

- **Examples (Numeric Mode):**

chmod 755 myscript.sh # rwxr-xr-x (Owner has read/write/execute, Group/Others have read/execute)

chmod 644 myfile.txt # rw-r--r-- (Owner has read/write, Group/Others have read only)

chmod 700 private_folder/ # rwx------ (Owner has full access, Group/Others have no access)

chmod 666 shared_file.txt # rw-rw-rw- (Everyone has read/write) - generally not recommended for security

chmod -R 775 my_project/ # Recursively set permissions of 'my_project' and its c

2.4 Common System Commands (Short Note)

This section covers basic commands for interacting with the Linux system, checking information, and managing your terminal.

1. **who**: Shows who is currently logged into the system, including their username, terminal, login time, and originating host.
    - **Example:** who
2. **whoami**: Displays your current effective username (i.e., who you are logged in as).
    - **Example:** whoami
3. **man (Manual)**: Provides detailed online reference manuals (man pages) for Linux commands, utilities, and other system components. Essential for learning command options and usage.
    - **Example:** man ls (to view the manual for ls)
4. **echo**: Prints a line of text or the value of variables to the terminal. Commonly used for displaying messages in scripts.
    - **Example:** echo "Hello Linux!" or echo $HOME
5. **date**: Displays the current system date and time. Can also be used to set the date/time (with root privileges) or format the output.
    - **Example:** date or date "+%Y-%m-%d"
6. **clear**: Clears the terminal screen, moving all previous output out of view to provide a clean prompt.

- **Example:** clear (or press Ctrl + L)

2.5 Text Processing Commands

These commands are essential for manipulating, analyzing, and transforming text data, which is a common task in Linux environments. They often work best when chained together using pipes (|).

1. head

- **Purpose:** The head command outputs the first part (beginning) of files to standard output. By default, it shows the first 10 lines.
- **Syntax:**

  head [options] [file(s)]
- **Common Options:**
  - -n NUM or --lines=NUM: Output the first NUM lines.
  - -c NUM or --bytes=NUM: Output the first NUM bytes.
  - -q or --quiet: Never print headers giving file names (useful when processing multiple files).
- **Examples:**

```
head myfile.txt              # Displays the first 10 lines of myfile.txt
head -n 5 access.log         # Displays the first 5 lines of access.log
head -c 200 document.txt     # Displays the first 200 bytes of document.txt
head -n 2 file1.txt file2.txt # Displays the first 2 lines of both file1.txt and file2.txt
```

2. tail

- **Purpose:** The tail command outputs the last part (end) of files to standard output. By default, it shows the last 10 lines. It's particularly useful for monitoring log files in real-time.
- **Syntax:**

  tail [options] [file(s)]
- **Common Options:**
  - -n NUM or --lines=NUM: Output the last NUM lines.

- -c NUM or --bytes=NUM: Output the last NUM bytes.
- -f or --follow: **Follow mode**. Outputs appended data as the file grows (great for monitoring live log files). Press Ctrl+C to exit.
- -q or --quiet: Never print headers giving file names.

- **Examples:**

tail mylog.log # Displays the last 10 lines of mylog.log

tail -n 20 error.log # Displays the last 20 lines of error.log

tail -f /var/log/syslog # Monitors the syslog file in real-time

tail -n +100 large_file.txt # Displays content starting from line 100 to the end

3. cut

- **Purpose:** The cut command extracts sections from each line of a file. It can cut based on byte position, character position, or field delimiter.
- **Syntax:**

cut [options] [file(s)]

- **Common Options:**
  - -f n: Select fields (columns) specified in LIST. Fields are typically separated by a delimiter. LIST can be single numbers (e.g., 1), ranges (e.g., 1-3), or combinations (e.g., 1,3,5).
  - -d (delimiter): Use DELIM instead of the default tab for field separation.
  - -c (characters: Select characters at positions specified in LIST.
  - -b (bytes): Select bytes at positions specified in LIST.
- **Examples:** Assume users.txt contains:
- john:x:1001:1001:John Doe:/home/john:/bin/bash
- mary:x:1002:1002:Mary Smith:/home/mary:/bin/sh

cut -d: -f1,5 users.txt # Extracts the first and fifth fields (username and full name), using ':' as delimiter

Output:

john:John Doe

mary:Mary Smith

cut -c 1-5,10 users.txt # Extracts characters 1-5 and the 10th character of each line

echo "apple orange banana" | cut -d' ' -f2 # Uses pipe, extracts the second word separated by space

Output: orange

4. sort

- **Purpose:** The sort command sorts lines of text files. It can sort alphabetically, numerically, by specific fields, and in various orders.
- **Syntax:**

sort [options] [file(s)]

- **Common Options:**
  - -r: Reverse the result of comparisons.
  - -n: Compare according to string numerical value.
  - -k Sort via a key. POS1 is the starting field/character, POS2 is the ending field/character.
  - -t: Use SEP as field separator.
  - -u or --unique: With -c, check for strict ordering; without -c, output only the first of a run of identical lines.
  - -o OUTPUT_FILE: Write the result to OUTPUT_FILE instead of standard output.

Examples: Assume names.txt contains:

  - Bob
  - Alice
  - Charlie
  - Eve
  - David

sort names.txt # Sorts alphabetically (Alice, Bob, Charlie, David, Eve)

sort -r names.txt # Sorts in reverse alphabetical order (Eve, David, ...)

sort -n numbers.txt # Sorts a file of numbers numerically (e.g., 1, 10, 2, 20 would become 1, 2, 10, 20)

sort -k 2 -t, data.csv # Sorts 'data.csv' based on the second field, using ',' as separator

cat access.log | sort | uniq # Sorts a log file and then removes duplicate lines

5. cmp (Compare)

- **Purpose:** The cmp command compares two files byte by byte and reports the first differing byte and line number.
- **Syntax:**

cmp [options] FILE1 FILE2

- **Output:**
  - If files are identical, cmp returns no output and an exit status of 0.
  - If files differ, it reports the byte and line number where the first difference occurs, and returns a non-zero exit status.
- **Common Options:**
  - -s or --quiet or --silent: Suppress all output. Only return exit status.
  - -l or --verbose: Output the byte number and the differing bytes (in octal) for all differences.
- **Examples:**

cmp file1.txt file2.txt # Compares file1.txt and file2.txt

cmp -s original.txt copy.txt # Compares silently; useful in scripts to check if files are identical

6. tr (Translate or Delete Characters)

- **Purpose:** The tr command translates or deletes characters from standard input and writes to standard output. It's often used with pipes.
- **Syntax:**

tr [options] SET1 [SET2]

- **Key Concepts:**
  - SET1: The set of characters to search for.
  - SET2: The set of characters to replace them with. If SET2 is shorter than SET1, SET1 characters at the end are deleted. If SET2 is longer, the extra characters are ignored.
- **Common Options:**

- -d or --delete: Delete characters in SET1.
- -s or --squeeze-repeats: Replace each sequence of a repeated character that is in SET1 with a single occurrence of that character.

- **Examples:**

echo "hello world" | tr '[:lower:]' '[:upper:]' # Converts lowercase to uppercase

Output: HELLO WORLD

echo "HeLlO wOrLd" | tr 'a-z' 'A-Z' # Same as above, using character ranges

echo "remove all spaces" | tr -d ' ' # Deletes all spaces

Output: removeallspaces

echo "helloooo worlddd" | tr -s 'o' 'd' # Squeezes repeated 'o's, then translates them to 'd'

Output: helod worldd

cat myfile.txt | tr -d '\r' > myfile_unix.txt # Removes Windows carriage returns ('\r') for Unix compatibility

7. uniq (Unique)

- **Purpose:** The uniq command reports or filters out repeated adjacent lines from a sorted file. It's crucial to understand that uniq only works on *consecutive* duplicate lines. Therefore, you almost always sort the file first.
- **Syntax:**

uniq [options] [input_file] [output_file]

- **Common Options:**
  - -c or --count: Prefix lines by the number of occurrences.
  - -d or --repeated: Only print duplicate lines.
  - -u or --unique: Only print unique lines (non-repeated lines).
- **Examples:** Assume data.txt contains (note: line 2 and 3 are identical and consecutive):

- apple
- banana
- banana
- orange

- apple
- grape

uniq data.txt # Output (apple, banana, orange, apple, grape - it misses the second 'apple')

sort data.txt | uniq # Correct way to get unique lines regardless of position

Output:
apple
banana
grape
orange

sort data.txt | uniq -c # Counts occurrences of each unique line

Output:
2 apple
2 banana
1 grape
1 orange

sort data.txt | uniq -d # Shows only the lines that appeared more than once (only the first instance)

Output:
apple
banana

8. wc (Word Count)

- **Purpose:** The wc command prints newline, word, and byte (or character) counts for each file.
- **Syntax:**

wc [options] [file(s)]

- **Common Options:**
  - -l or --lines: Print the newline counts only.
  - -w or --words: Print the word counts only.
  - -c or --bytes: Print the byte counts only.
  - -m or --chars: Print the character counts only (takes multi-byte characters into account).

- **Examples:**

Bash

```
wc myfile.txt              # Prints lines, words, and bytes for myfile.txt
                           # Output: 15  120 800 myfile.txt (15 lines, 120 words, 800 bytes)
wc -l document.log         # Prints only the number of lines in document.log
ls -l | wc -l              # Counts the number of files/directories in the current directory
echo "Count these words" | wc -w # Counts words from piped input
                           # Output: 3
```

9. tee

- **Purpose:** The tee command reads from standard input and writes to both standard output (the screen) and one or more files. It's like a T-junction in a pipeline, allowing you to see the output while also saving it to a file.
- **Syntax:**

Bash

```
tee [options] [file(s)]
```

- **Common Options:**
  - -a or --append: Append output to the given files, rather than overwriting them.
  - -i or --ignore-interrupts: Ignore interrupt signals (like Ctrl+C).
- **Examples:**

Bash

```
ls -l | tee file_list.txt   # Displays the directory listing on screen AND saves it to file_list.txt
```

2.6 Introduction to Process

In Linux (and other Unix-like operating systems), a **process** is an instance of a running program. When you execute a command or run a script, the kernel creates a new process for it. Each process has its own unique set of resources and its own distinct environment.

Key Concepts of a Process:

1. **Process ID (PID):** Every process is assigned a unique positive integer called its Process ID. This ID is used by the kernel to keep track of and manage the process.

2. **Parent Process ID (PPID):** Most processes are started by another process. The process that starts another process is called its "parent." The child process will have a PPID that points to its parent's PID. The first process started by the kernel (usually systemd or init) has a PID of 1 and no parent.
3. **User ID (UID) & Group ID (GID):** Every process runs with the permissions of a specific user and group. These determine what files and resources the process can access.
4. **State:** A process can be in various states:
    - **Running (R):** The process is currently executing or ready to execute.
    - **Sleeping (S):** The process is waiting for an event (e.g., I/O to complete, a signal, a specific time).
    - **Stopped (T):** The process has been suspended, usually by a user (e.g., Ctrl+Z) or a signal. It can be resumed.
    - **Zombie (Z):** A child process that has terminated but whose entry still exists in the process table because its parent hasn't yet read its exit status. Zombie processes consume very little system resources but can indicate a programming error if they accumulate.
    - **Defunct (D):** Uninterruptible sleep, usually waiting for I/O.
5. **Priority:** Processes have a priority (or "nice value") that influences how much CPU time they get relative to other processes. Higher priority means more CPU time.
6. **Memory:** Each process has its own dedicated memory space, preventing one process from accidentally corrupting another's memory.
7. **Open Files:** A process keeps track of all the files it has open for reading, writing, or execution.

Process Lifecycle (Simplified):

1. **Creation (Fork):** A new process is created, typically by a parent process calling the fork() system call. The child process is initially a copy of the parent.
2. **Execution (Exec):** The child process then calls exec() to load a new program into its memory space and start executing it.
3. **Termination (Exit):** A process terminates when it finishes its task, encounters an error, or receives a signal to stop. It returns an exit status to its parent.
4. **Wait:** The parent process typically calls wait() to collect the exit status of its child process. If the parent doesn't wait(), the child becomes a zombie.

Foreground vs. Background Processes:

- **Foreground Process:** A process that is directly interacting with the user via the terminal. It waits for user input and sends its output directly to the terminal. You cannot type new commands while a foreground process is running (unless it's waiting for input).
- **Background Process:** A process that runs independently of the terminal's input. It generally does not wait for user input and can be run while you continue to type other commands. Its output might still appear on the terminal unless redirected.

2.7 Process Control Commands

These commands allow you to view, manage, and terminate processes on your system.

1. ps (Process Status)

- **Purpose:** The ps command reports a snapshot of the current processes. It displays information about processes running on your system.
- **Syntax:**

Bash

ps [options]

- **Key Concept:** ps shows processes that were running *at the moment* the command was executed. For real-time, continuous monitoring, top or htop are better.
- **Common Options (often combined):**
  - a: Display processes of all users on the terminal.
  - u: Display user-oriented format (UID, PID, PPID, CPU %, Mem %, Start Time, TTY, CMD).
  - x: Display processes without a controlling TTY (usually daemon processes).
  - e: Select all processes (long format).
  - f: Full format listing (more details).
  - l: Long format listing.
  - aux (common combination): Displays all user processes, with user/CPU/memory usage, and full command.
  - -ef (common combination): Displays all processes in full format.
  - -p PID: Display information for a specific PID.

- --forest: Show process parent/child relationships in a tree format.

- **Output Columns (common with ps aux):**
  - USER: The effective user ID of the process owner.
  - PID: Process ID.
  - %CPU: CPU usage of the process.
  - %MEM: Memory usage of the process.
  - VSZ: Virtual memory size in kilobytes.
  - RSS: Resident Set Size (physical memory used) in kilobytes.
  - TTY: Controlling terminal.
  - STAT: Process state (R=running, S=sleeping, T=stopped, Z=zombie,
  - START: Start time of the process.
  - TIME: Total CPU time used by the process.
  - COMMAND: The command that started the process.

- **Examples:**

Bash

```
ps                  # Shows processes associated with your current terminal
ps aux              # Shows all processes from all users, with detailed info
ps -ef              # Another common way to see all processes with full details
ps -p 1234          # Displays information about process with PID 1234
ps aux | grep firefox    # Finds all processes related to Firefox
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head -n 10 # Custom output, top 10 by memory
```

2. fg (Foreground)

- **Purpose:** The fg command resumes a job from the background and brings it back to the foreground, allowing it to interact with the terminal again.

- **Syntax:**

Bash

```
fg [job_ID]
```

- **Key Concept:** When you start a process and then suspend it using Ctrl+Z, it becomes a stopped job. fg can bring it back. If there's only one background job, you don't need to specify job_ID. You can see background jobs using the jobs command.
- **Examples:**

Bash

```
# 1. Start a command that runs indefinitely or takes time
ping google.com


# 2. Suspend it by pressing Ctrl+Z
# Output: [1]+  Stopped                 ping google.com


# 3. Check its status
jobs
# Output: [1]+  Stopped                 ping google.com


# 4. Bring it back to the foreground
fg 1                # Or simply 'fg' if it's the only job
# The 'ping' command resumes, and you regain control of the terminal.
```

- **Use Case:** Resuming a paused task, bringing a background process to interact with it.

3. bg (Background)

- **Purpose:** The bg command resumes a suspended job in the background. The job will continue to run but will not interact with the terminal (unless its output is redirected).
- **Syntax:**

Bash

```
bg [job_ID]
```

- **Key Concept:** Similar to fg, it works on jobs previously suspended with Ctrl+Z. If you want to start a process directly in the background, append & to the command (e.g., command &).
- **Examples:**

Bash

```
# 1. Start a command and suspend it (Ctrl+Z), as in the `fg` example
ping google.com
# Press Ctrl+Z
# Output: [1]+  Stopped                 ping google.com


# 2. Put it in the background
bg 1                # Or simply 'bg' if it's the only job
```

4. kill

- **Purpose:** The kill command sends a signal to a process, typically to terminate it.
- **Syntax:**

Bash

```
kill [options] PID
kill -SIGNAL PID
kill -SIGNAL %job_ID
```

- **Key Concepts:**
  - **Signals:** kill doesn't "kill" in the sense of instant death. It sends a *signal* to a process. The process decides how to handle that signal (though some signals cannot be ignored).
  - **Common Signals:**
    - **TERM (15):** The default signal. Requests the process to terminate gracefully (allows it to clean up). Equivalent to kill PID.
    - **KILL (9):** A forceful termination signal. The process cannot ignore this and is immediately killed. Use this as a last resort if TERM doesn't work. Equivalent to kill -9 PID.
    - **HUP (1):** Hang Up. Often used to tell a daemon process to reload its configuration files without restarting.
    - **STOP (19):** Stop a process (like Ctrl+Z), but doesn't get delivered to the terminal.

        - **CONT (18):** Continue a stopped process.
    - You can specify signal names (e.g., KILL) or their numeric values (e.g., 9).
- **Examples:**

Bash

```
# Find a process ID first (e.g., of a hung program)
ps aux | grep "hung_program"
# Output: user    1234  0.5  2.0 ... /usr/bin/hung_program

kill 1234               # Sends SIGTERM (15) to PID 1234 (graceful termination attempt)
kill -9 1234            # Sends SIGKILL (9) to PID 1234 (forceful termination)
kill -HUP 5678          # Sends SIGHUP (1) to PID 5678 (often for config reload)

# You can also use job IDs (from the `jobs` command)
jobs
# Output: [1]+  Stopped                 long_running_script.sh
kill %1                 # Sends SIGTERM to job 1
```

- **Use Case:** Terminating unresponsive programs, gracefully restarting services, or sending specific signals to processes.

5. sleep

- **Purpose:** The sleep command pauses execution for a specified amount of time.
- **Syntax:**

Bash

```
sleep NUMBER[SUFFIX]
```

- **Key Concepts:**
    - NUMBER: The duration.
    - SUFFIX: Optional.
        - s: seconds (default if no suffix)

- m: minutes
- h: hours
- d: days

- **Examples:**

Bash

```
echo "Starting in 5 seconds..."
sleep 5                 # Pauses for 5 seconds
echo "Done!"

sleep 10m               # Pauses for 10 minutes
sleep 2h                # Pauses for 2 hours

# In a script for a retry mechanism:
#!/bin/bash
echo "Attempting connection..."
if ! some_command; then
  echo "Connection failed. Retrying in 10 seconds..."
  sleep 10
  some_command
fi
```

2.8 Job Scheduling Commands

These commands allow you to schedule tasks to run at a later time, either once or repeatedly.

1. at

- **Purpose:** The at command schedules commands to be executed **once** at a specified time.
- **Syntax:**

Bash

at [options] TIME

- **Key Concepts:**
  - TIME can be very flexible (e.g., now + 10 minutes, tomorrow 08:00 AM, 14:30, noon, midnight, teatime).
  - After typing at TIME, the prompt changes to at>, where you type the commands you want to execute. Press Ctrl+D on a new line to finish and schedule the job.
  - atq (or at -l): Lists pending at jobs.
  - atrm JOB_ID (or at -d JOB_ID): Deletes a pending at job.
- **Examples:**

Bash

at now + 10 minutes # Schedule a job to run in 10 minutes

> echo "Hello from the future!" > /tmp/future_message.txt

> Ctrl+D

job 5 at 2025-07-08 09:50

at 14:30 tomorrow # Schedule a job for 2:30 PM tomorrow

> /home/user/my_script.sh

> Ctrl+D

At q # List pending 'at' jobs

Output:

5 2025-07-08 09:50 a root

6 2025-07-09 14:30 a root

At rm 5 # Delete job with ID 5

2. batch

- **Purpose:** The batch command is similar to at but schedules commands to be executed **when the system load average permits**. It means the commands will run only when the system is relatively idle.
- **Syntax:**

Bash

batch

- **Key Concept:** It implicitly uses at now, but the execution is delayed until the system's load average drops below a certain threshold (typically 0.8, but configurable). This is ideal for non-urgent, resource-intensive tasks.
- **Examples:**

Bash

batch # Enter commands to be executed when system load is low

> tar -czf /backup/daily_backup.tar.gz /var/www/html

> Ctrl+D

job 7 at 2025-07-08 09:40

The job is scheduled immediately, but won't run until the system is idle.

- **Use Case:** Running computationally heavy tasks like backups, data processing, or large compilations without impacting interactive user performance during busy periods.

3. crontab

- **Purpose:** The crontab (cron table) command is used to schedule commands to be executed **periodically** (repeatedly) at fixed times, dates, or intervals. These are known as "cron jobs."
- **Syntax:**

Bash

crontab [options]

- **Key Concepts:**

- Each user has their own crontab file, which is a plain text file containing cron job entries.
- The crontab syntax uses five fields to define the schedule, followed by the command to execute:
- * * * * * command_to_execute
- | | | | |
- | | | | ----- Day of week (0 - 7, Sunday is 0 or 7)
- | | | ------- Month (1 - 12)
- | | --------- Day of month (1 - 31)
- | ----------- Hour (0 - 23)
- ------------ Minute (0 - 59)
- An asterisk (*) means "every" (e.g., * in the minute field means "every minute").
- Ranges (e.g., 1-5), lists (e.g., 1,3,5), and step values (e.g., */15 for every 15 minutes) are supported.
- **Environment:** Cron jobs run in a minimal environment. It's often best to use absolute paths for commands and scripts, and explicitly set environment variables if needed within the script itself.
- **Output:** Any output from a cron job (stdout or stderr) is typically emailed to the user who owns the crontab. You can redirect output to /dev/null if you don't want emails.

- **Common Options:**
  - -e: **Edit** the user's crontab file (opens in your default text editor).
  - -l: **List** the user's current crontab entries.
  - -r: **Remove** the user's current crontab file (deletes all cron jobs for that user).
  - -v: View the last time crontab was edited.
- **Examples:**

Bash

```
crontab -e          # Opens your personal crontab for editing
```