Linux OS

Date Created	@November 3, 2022 11:12 AM
Progress?	Done - need to make it pretty
Done	Done
Status	☑

Linux Command line
  Navigation and File Management
  Text Editing
  Input/Output
  Text Processing
  Process Management
  System Information
  User Management
  Network Management
  Compression and Archiving
  System Administration
  Miscellaneous

Linux Command line

▼ Navigation and File Management

▼ pwd - print working directory

The `pwd` command is a basic command in Linux and Unix-like operating systems that is used to print the current working directory's absolute path.

▼ cd - Change working directory.

The `cd` command is a fundamental command in Linux and Unix-like operating systems that is used to change the current working directory. When a user logs in to a Linux system, the current working directory is the **home directory of that user**. However, users can use the `cd` command to navigate to other directories.

The syntax of the `cd` command is simple. The user needs to type `cd` followed by the path of the directory to which they want to navigate. The path can be either an absolute path or a relative path.

An absolute path is a path that starts from the root directory (`/`), which is the top-level directory of the file system. For example, the absolute path of the home directory of the user `john` on a Linux system could be `/home/john`.

A relative path is a path that starts from the current working directory. For example, if the current working directory is `/home/john`, and the user wants to navigate to the `Documents` directory, they can use the relative path `cd Documents`.

Here are some examples of how to use the `cd` command:

- `cd` or `cd ~`: Change to the home directory of the current user.
- `cd /`: Change to the root directory.
- `cd /home`: Change to the `/home` directory.
- `cd ..`: Change to the parent directory of the current directory.
- `cd -`: Change to the previous working directory.
- `cd ./directory` or `cd directory`: Change to a directory that is located in the current directory.
- `cd /directory`: Change to a directory that is located at the root of the file system.
- `cd ~/directory`: Change to a directory that is located in the home directory of the current user.

It's important to note that the `cd` command is **case-sensitive,** so if the directory name is capitalized, the command must be capitalized as well. Additionally, if the directory name contains spaces or special characters, it must be enclosed in quotes or escaped with a backslash.

Finally, it's worth noting that the `cd` command only changes the working directory for the current shell session. If a new shell is opened, the working directory will be the same as it was before the `cd` command was executed.

ls - List directory contents.

The `ls` command is a basic command in Linux and Unix-like operating systems that is used to **list the contents of a directory.** By default, when a user enters the `ls` command without any arguments, it lists the **contents** of the current working directory.

Here are some examples of how to use the `ls` command:

- `ls`: List the contents of the current working directory.
- `ls -l`: List the contents of the current working directory in **long format**.
- `ls -a`: List all files and directories, including **hidden files** and directories.
- `ls -lh`: List the contents of the current working directory in long format with **human-readable file sizes.**
- `ls -R`: List the contents of the current working directory and all subdirectories recursively.
    - When the -R option is used with the ls command, it tells the command to recursively list the contents of the current working directory and any subdirectories. This means that the ls command will not only list the files and directories in the current working directory but also the files and directories in all the subdirectories of the current working directory, and any subdirectories within those subdirectories.
- `ls -t`: List the contents of the current working directory sorted by **modification time.**
- `ls -S`: List the contents of the current working directory **sorted by file size.**
- `ls -r`: List the contents of the current working directory in **reverse order.**

The `ls` command can also be used with file or directory names as arguments to list the contents of a specific directory. For example, `ls /home/john/Documents` will list the contents of the `Documents` directory of the user `john`.

Some additional options that can be used with the `ls` command include:

- `d`: List only the directory names, not their contents.
- `i`: Display the inode number of each file or directory.
- `L`: Follow symbolic links to their target directories or files.
- `F`: Append a character to each file name to indicate its type (`/` for directories, ` ` for executables, `@` for symbolic links).
- `G`: Display the output in color.

It's worth noting that the `ls` command is case-sensitive, so if a file or directory name is capitalized, the command must be capitalized as well. Additionally, if a file or directory name contains spaces or special characters, it must be enclosed in quotes or escaped with a backslash.

Overall, the `ls` command is a fundamental command in Linux that is used to display the contents of a directory, and it offers a variety of options to customize the output.

▼ mkdir - Create a new directory.

The `mkdir` command is a basic command in Linux and Unix-like operating systems that is used to create a new directory. The `mkdir` command stands for "make directory."

Here are some examples of how to use the `mkdir` command:

- `mkdir directory`: Create a new directory called `directory` in the current working directory.
- `mkdir /home/john/Documents/new_directory`: Create a new directory called `new_directory` in the `Documents` directory of the user `john`.
- `mkdir -p /home/john/Documents/new_directory/subdirectory`: Create a new directory called `new_directory` in the `Documents` directory of the user `john`, along with a subdirectory called `subdirectory`.

  The `-p` option used in this example creates any necessary parent directories **that do not exist**. If the `Documents` directory does not exist, it will be created along with the `new_directory` and `subdirectory` directories.

When a new directory is created using the `mkdir` command, the **user who created the directory becomes the owner of the directory,** and the directory's permissions are set to the **default permissions** for newly created directories.

It's important to note that the `mkdir` command can only **create directories and cannot create files.** Additionally, if a directory with the same name already exists in the current working directory or in the specified path, the `mkdir` command will not create a new directory and will instead display an error message.

Finally, it's worth noting that the `mkdir` command is a built-in command in most Linux and Unix-like operating systems, which means that it's part of the shell itself and doesn't require an external program to be executed.

▼ rmdir - Remove an empty directory.

The `rmdir` command is a basic command in Linux and Unix-like operating systems that is used to remove an **empty** directory. The `rmdir` command stands for "remove directory."

Here are some examples of how to use the `rmdir` command:

- `rmdir directory`: Remove the directory called `directory` in the current working directory.
- `rmdir /home/john/Documents/new_directory`: Remove the directory called `new_directory` in the `Documents` directory of the user `john`.

The `rmdir` command **can only remove empty directories**. If the directory contains files or other directories, the `rmdir` command will not remove the directory, and an error message will be displayed.

It's also worth noting that the `rmdir` command can only remove empty directories. If the directory contains files or other directories, the `rmdir` command will not remove the directory, and an error message will be displayed. **In this case, the `rm` command can be used to remove the directory and its contents recursively.**

Finally, it's worth noting that the `rmdir` command is a built-in command in most Linux and Unix-like operating systems, which means that it's part of the shell itself and doesn't require an external program to be executed.

▼ touch - Create an empty file or update the timestamp of an existing file.

The `touch` command is a basic command in Linux and Unix-like operating systems that is used to create a new file or update the modification and access time of an existing file. The `touch` command is also used to change the **timestamps of a file.**

Here are some examples of how to use the `touch` command:

- `touch file.txt`: Create a new file called `file.txt` in the current working directory with the current timestamp.
- `touch /home/john/Documents/file.txt`: Create a new file called `file.txt` in the `Documents` directory of the user `john` with the current timestamp.
- `touch -d "2022-12-31" file.txt`: Change the modification and access time of the file `file.txt` to December 31, 2022.
- `touch -t 202212310000.00 file.txt`: Change the modification and access time of the file `file.txt` to December 31, 2022, at 00:00:00.

When a new file is created using the `touch` command, an empty file with the specified name is created in the current working directory or in the specified path. **If the file already exists, the `touch` command will update the modification and access time of the file to the current time by default.**

The `-d` option used in the third example changes the modification and access time of the file to the specified date and time. The date must be in the format `YYYY-MM-DD`, and the time must be in the format `HH:MM:SS`.

The `-t` option used in the fourth example changes the modification and access time of the file to the specified date and time. The date must be in the format `YYYYMMDD`, and the time must be in the format `HHMM.SS`.

It's important to note that the `touch` command does not change the content of a file, only the timestamps. If a file does not exist and is created using the `touch` command, the file will be empty.

▼ cp - Copy files or directories.

The `cp` command is a basic command in Linux and Unix-like operating systems that is used to copy **files or directories** from one location to another. The `cp` command stands for "copy."

Here are some examples of how to use the `cp` command:

- `cp file.txt /home/john/Documents/`: Copy the file `file.txt` from the current working directory to the `Documents` directory of the user `john`.
- `cp -r directory /home/john/Documents/`: Copy the directory `directory` from the current working directory to the `Documents` directory of the user `john` **recursively**.
- `cp -i file.txt backup.txt`: Copy the file `file.txt` to a new file called `backup.txt`, but prompt the user for confirmation if the destination file already exists.

The `-r` option used in the second example copies the directory `directory` and all its contents recursively. If the directory contains subdirectories, the `cp` command will copy them as well.

The `-i` option used in the third example prompts the user for confirmation if the destination file already exists. This option can be useful to avoid accidentally overwriting an existing file.

It's important to note that the `cp` command can only copy files and directories that the user has **permission to access.** Additionally, if a file or directory with the same name already exists in the destination directory, the `cp` command will ***overwrite*** it by default, unless the `-i` option is used.

Finally, it's worth noting that the `cp` command is a built-in command in most Linux and Unix-like operating systems, which means that it's part of the shell itself and doesn't require an external program to be executed.

Overall, the `cp` command is a simple but essential command in Linux that is used to copy files and directories from one location to another.

▼ mv - Move or rename files or directories.

The `mv` command is a basic command in Linux and Unix-like operating systems that is used to ***move or rename*** files and directories. The `mv` command stands for "move."

Here are some examples of how to use the `mv` command:

- `mv file.txt /home/john/Documents/`: Move the file `file.txt` from the current working directory to the `Documents` directory of the user `john`.
- `mv directory /home/john/Documents/new_directory`: Move the directory `directory` from the current working directory to the `Documents` directory of the user `john` and **rename** it to `new_directory`.
- `mv file.txt backup.txt`: **Rename** the file `file.txt` to `backup.txt` in the current working directory.

When a file or directory is moved using the `mv` command, it is essentially being renamed or transferred to a new location. If the file or directory is moved to a different file system or partition, it is copied to the new location and then deleted from the original location.

It's important to note that the `mv` command can only move files and directories that the user **has permission to access**. Additionally, if a file or directory with the same name already exists in the destination directory, the `mv` command will **overwrite it by default.**

Finally, it's worth noting that the `mv` command is a built-in command in most Linux and Unix-like operating systems, which means that it's part of the shell itself and doesn't require an external program to be executed.

Overall, the `mv` command is a simple but essential command in Linux that is used to move or rename files and directories.

▼ rm - Remove files or directories.

The `rm` command is a basic command in Linux and Unix-like operating systems that is used to remove **files and directories.** The `rm` command stands for "remove."

Here are some examples of how to use the `rm` command:

- `rm file.txt`: Remove the file `file.txt` from the current working directory.
- `rm -r directory`: Remove the directory `directory` from the current working directory **and all its contents recursively.**
- `rm -i file.txt`: Remove the file `file.txt`, but prompt the user for confirmation before actually removing it.

The `-r` option used in the second example removes the directory `directory` and all its contents recursively. If the directory contains subdirectories, the `rm` command will remove them as well.

The `-i` option used in the third example prompts the user for confirmation before actually removing the file. This option can be useful to avoid accidentally deleting an important file.

It's important to note that the `rm` command is a potentially dangerous command since it permanently deletes files and directories and their contents. Therefore, it's important to use the `rm` command with caution and to double-check that the correct file or directory is being removed before executing the command.

Additionally, the `rm` command can only remove files and directories that the user has **permission to access**. If a file or directory is write-protected or owned by another user, the `rm` command will not be able to remove it.

Finally, it's worth noting that the `rm` command is a built-in command in most Linux and Unix-like operating systems, which means that it's part of the shell itself and doesn't require an external program to be executed.

Overall, the `rm` command is a simple but potentially dangerous command in Linux that is used to remove files and directories.

▼ Text Editing

▼ what are the basic text editors in linux?

There are several text editors available in Linux, ranging from simple and user-friendly editors to more advanced and powerful editors. Here are some of the most commonly used basic text editors in Linux:

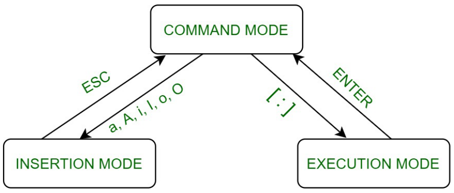
1. **Nano**: A simple and user-friendly text editor that is designed to be easy to use and require minimal learning.
2. **Vi**: A more advanced text editor that has been around since the early days of Unix. It is known for its powerful features and keyboard shortcuts.
3. **Vim**: A modern version of Vi that has additional features and improvements. It is often used by advanced users who need a powerful text editor.
4. **Emacs**: A highly customizable text editor that is designed to be extensible and flexible. It is often used by developers and programmers who need a text editor with advanced features.

▼ Vim Editor

▼ Basics

Here are the key points about Vim for beginners:

- Vim has two modes: **command mode and insert mode**. Command mode is for moving around and issuing commands. Insert mode is for inserting text.



- To enter insert mode, press i. To exit insert mode, press Esc.
- In command mode, you can use arrow keys or h, j, k, l (left, down, up, right) to move the cursor.

- To edit text, enter insert mode, edit the text, then exit insert mode (with Esc).

▼ Commands

▼ Navigation Commands:

- `h` , `j` , `k` , `l` : Move cursor left, down, up, and right respectively
- `0` : Move cursor to the start of the line
- `$` : Move cursor to the end of the line
- `gg` : Move cursor to the beginning of the file
- `G` : Move cursor to the end of the file
- `Ctrl+f` : Scroll forward one page
- `Ctrl+b` : Scroll backward one page

▼ Editing Commands:

- `i` : This key enters Insert mode at the **current** cursor position.
- `a` : This key enters Insert mode **after the current** cursor position.
- `o` : This key inserts a **new line below** the current line and enters Insert mode.
- `O` : This key inserts a **new line above** the current line and enters Insert mode.
- `x` : This key **deletes** the **character** under the cursor.
- `dd` : This key **deletes the current line.**
- `yy` : This key **copies** the current line.
- `p` : This key pastes the copied or cut text after the cursor.
- `u` : This key **undoes** the last change.
- `Ctrl+r` : This key **redoes** the last change.

▼ Search and Substitute Commands:

▼ Search:

- `/pattern` : This key searches for a pattern in the file.
    - `n` : This key moves the cursor to the next search result.
    - `N` : This key moves the cursor to the previous search result.

▼ Substitute or Replace:

The substitute command in Vim is a powerful command that is used to **search for and replace text** in a file. Here is an exhaustive explanation of the substitute command in Vim editor with all variations:

▼ Basic syntax:

```
:%s/old-text/new-text/g
```

This command searches for **all occurrences** of "old-text" in the entire file and replaces them with "new-text". The "g" flag at the end of the command stands for **global**, which means to replace all occurrences of "old-text" in the file.

- %s - indicates which line number
- /g - replace all the “old” in all words inside the line selected **i.e line type**

    - **globally (on all occurrences of the pattern).**
    - /1 - replace only the first occurrence of the word in the lines selected

  For example, the following command replaces all occurrences of the word "apple" with the word "orange" in all lines in the entire file:

```
:%s/apple/orange/g
```

▼ **Replace only in certain lines:**

```
:line-number s/old-text/new-text/g
```

This command searches for all occurrences of "old-text" in the specified line number and replaces them with "new-text".

  For example, the following command replaces all occurrences of the word "apple" with the word "orange" in line 10:

```
:10s/apple/orange/g
```

▼ **Search and replace in a range of lines:**

```
:start-line, end-line s/old-text/new-text/g
```

This command searches for all occurrences of "old-text" in the range of lines from "start-line" to "end-line" and replaces them with "new-text".

  For example, the following command replaces all occurrences of the word "apple" with the word "orange" in lines 5 to 10:

```
:5,10s/apple/orange/g
```

▼ **Replace only the first *occurrence* of a word in each line:**

```
:%s/old-text/new-text/
```

This command searches for the first occurrence of "old-text" in **each line** and replaces it with "new-text". The "g" flag is not present, so only the first occurrence of "old-text" **in each line is replaced.**

  For example, the following command replaces the first occurrence of the word "apple" with the word "orange" in the entire file but **only the first *occurrence***:

```
:%s/apple/orange/
```

▼ **Replace with confirmation:**

```
:%s/old-text/new-text/gc
```

This command searches for all occurrences of "old-text" in the entire file and replaces them with "new-text", but asks for confirmation before each replacement. The "c" flag at the end of the command stands for "confirm".

For example, the following command replaces all occurrences of the word "apple" with the word "orange" in the entire file with confirmation:

```
:%s/apple/orange/gc
```

▼ Replace in a visual selection:

If you select a block of text in visual mode, then press the colon key, Vim will automatically enter the command line with the range for the current selection. You can then enter a substitute command to replace the selected text.

For example, if you select a block of text and then press the colon key, Vim will show the following command:

```
:'<,'>
```

You can then type a substitute command to replace the selected text. For example:

```
:'<,'>s/apple/orange/gc
```

This command searches for all occurrences of "apple" in the selected text and replaces them with "orange", but asks for confirmation before each replacement.

These are some of the most commonly used variations of the substitute command in Vim editor. The substitute command is a powerful tool that can help you quickly search for and replace text in a file, making editing and programming more efficient.

▼ Saving and Quitting Commands:

- `:w`: This key saves i.e **w**rites changes to the file.
- `:wq`: This key saves changes to the file and quits Vim.
- `:q`: This key quits Vim.
- `:q!`: This key quits Vim without saving changes.

▼ Input/Output

▼ cat - Concatenate and display files.

The `cat` command in Linux is a tool that is used to **concatenate** (means to join or link things together in a series.) files and **display** their contents. Here is an explanation of the `cat` command with all its variations and examples:

▼ Basic Syntax:

```
cat file1
```

This command concatenates and displays the contents of the specified files.

For example, the following command displays the contents of the file `example.txt`:

```
cat example.txt
```

▼ Display multiple files:

```
cat file1 file2 file3 ...
```

This command concatenates and displays the contents of the specified files. For example, the following command displays the contents of the files `file1.txt`, `file2.txt`, and `file3.txt`:

```
cat file1.txt file2.txt file3.txt
```

▼ Display a file with line numbers:

```
cat -n file
```

This command displays the contents of the specified file with line numbers.

▼ Display a file with line numbers but no blank lines:

```
cat -b file
```

This command displays the contents of the specified file with line numbers, but does not include line numbers for blank lines.

▼ Display a file with a specified number of lines:

```
cat -n file | head -n num
```

This command displays the specified number of lines from the beginning of the file with line numbers. For example, the following command displays the first 10 lines of the file `example.txt` with line numbers:

```
cat -n example.txt | head -n 10
```

▼ Append to a file:

```
cat file1 >> file2
```

This command appends the contents of `file1` to `file2`. For example, the command appends the contents of `file1.txt` to `file2.txt`:

▼ Create a new file:

```
cat > file
```

This command creates a new file and allows you to enter text into it. Press `Ctrl+D` when you are finished entering text to save the file. For example, the following command creates a new file called `example.txt`:

```
cat > example.txt
```

- **Display non-printable characters:**

  ```
  cat -v file
  ```

  This command displays the contents of the specified file, showing non-printable characters as visible characters. For example, the following command displays the contents of the file `example.txt`, showing non-printable characters as visible characters:

  ```
  cat -v example.txt
  ```

  examples of non-printable characters include the carriage return character "\r", which moves the cursor to the beginning of a line, and the newline character "\n", which creates a new line in the text.

echo - Display a message on the screen

The `echo` command in Linux is a tool that is used to **display text or variables** on the command line. Here is an explanation of the `echo` command with all its variations and examples:

- **Basic Syntax:**

  ```
  echo "text"
  ```

  This command displays the specified text on the command line. For example, the following command displays the text "Hello, World!":

  ```
  echo "Hello, World!"
  ```

- **Display the value of a variable:**

  ```
  echo $variable
  ```

  This command displays the value of the specified variable on the command line. For example, if the variable `name` is set to "John", the following command displays the text "John":

  ```
  name="John"
  echo $name
  ```

- **Display the output of a command:**

  ```
  echo $(command)
  ```

  This command displays the output of the specified command on the command line. For example, the following command displays the contents of the file `example.txt`:

  ```
  echo $(cat example.txt)
  ```

- **Display text without a newline [i.e in a paragraph manner]:**

  ```
  echo -n "text"
  ```

  This command displays the specified text on the command line without a newline character. For example, the following command displays the text "Hello, World!" without a newline:

```
echo -n "Hello, World!"
```

▼ Display text with a specific color:

```
echo -e "\033[color;message"
```

This command displays the specified text on the command line with a specific color. The `color` parameter specifies the color code, which can be one of the following:

- 0 for black, 1 for red, 2 for green ,3 for yellow ,4 for blue ,5 for magenta ,6 for cyan ,7 for white

The `message` parameter specifies the text to be displayed. For example, the following command displays the text "Hello, World!" in blue:

```
echo -e "\033[34mHello, World!\033[0m"
```

▼ Display a horizontal line:

```
echo "-------------------------"
```

These are some of the most commonly used variations of the `echo` command in Linux. The `echo` command is a simple but powerful tool that can help you quickly display text or variables on the command line, and can be useful in scripting and automation tasks.

▼ head - display the first few lines of a file

The `head` command in Linux is a tool that is used to display the first few lines of a file. Here is an explanation of the `head` command with all its variations and most used examples:

▼ Basic Syntax:

```
head file
```

This command displays the ***first 10 lines*** of the specified file. For example, the following command displays the first 10 lines of the file `example.txt`:

```
head example.txt
```

▼ Display a specific number of lines:

```
head -n num file
```

This command displays the specified number of lines from the **beginning of the file.** For example, the following command displays the first 5 lines of the file `example.txt`:

```
head -n 5 example.txt
```

▼ Display a file with line numbers:

```
head -n num file | cat -n
```

This command displays the specified number of lines from the beginning of the file with line numbers. The `head` command is used to display the specified number of lines, and the `cat -n` command is used to add line numbers to the output.

▼ Display a file with line numbers but no blank lines:

```
head -n num file | cat -b
```

This command displays the specified number of lines from the beginning of the file with line numbers, but does not include line numbers for blank lines. The `head` command is used to display the specified number of lines, and the `cat -b` command is used to add line numbers to the output, without including line numbers for blank lines.

▼ Display multiple files:

```
head file1 file2 file3 ...
```

This command displays the first 10 lines of each specified file. For example, the following command displays the first 10 lines of the files `file1.txt`, `file2.txt`, and `file3.txt`:

```
head file1.txt file2.txt file3.txt
```

▼ tail - display the last few lines of a file

The `tail` command in Linux is a tool that is used to display the **last few lines of a file**. Here is an exhaustive explanation of the `tail` command with all its variations and most used examples:

▼ Basic Syntax:

```
tail file
```

This command displays the **last 10 lines** of the specified file. For example, the following command displays the last 10 lines of the file `example.txt`:

```
tail example.txt
```

▼ Display a specific number of lines:

```
tail -n num file
```

This command displays the specified number of lines from the end of the file. For example, the following command displays the last 5 lines of the file `example.txt`:

```
tail -n 5 example.txt
```

▼ Display a file continuously:

```
tail -f file
```

This command displays the contents of the specified file continuously, as new data is added to the file. **This can be useful for monitoring log files or other continuously updated files**. For example, the following command displays the contents of the file `example.log` continuously:

```
tail -f example.log
```

▼ **Display a file with line numbers:**

```
tail -n num file | cat -n
```

This command displays the specified number of lines from the end of the file with line numbers. The `tail` command is used to display the specified number of lines, and the `cat -n` command is used to add line numbers to the output.

▼ **Display a file with line numbers but no blank lines:**

```
tail -n num file | cat -b
```

This command displays the specified number of lines from the end of the file with line numbers, but does not include line numbers for blank lines. The `tail` command is used to display the specified number of lines, and the `cat -b` command is used to add line numbers to the output, without including line numbers for blank lines.

▼ **Display multiple files:**

```
tail file1 file2 file3 ...
```

This command displays the last 10 lines of each specified file.

▼ more - displays the contents of the specified file *page by page* [only forward navigation]

The `more` command in Linux is a tool that is used to display the contents of a file page by page. Here is an exhaustive explanation of the `more` command with all its variations and most used examples:

▼ **Basic Syntax:**

```
more file
```

This command displays the contents of the specified file page by page. You can navigate through the pages by pressing the **spacebar to move forward one page,** and the "q" key to quit. For example, the following command displays the contents of the file `example.txt` page by page:

```
more example.txt
```

▼ **Display a file with line numbers:**

```
cat -n file | more
```

This command displays the contents of the file with line numbers and then shows the output page by page. The `cat -n` command is used to display the file with line numbers, and the `more` command is used to show the output page by page. For example, the following command displays the contents of the file `example.txt` with line numbers:

```
cat -n example.txt | more
```

- **Search for a pattern in a file:**

```
more file
/pattern
```

This command displays the contents of the file page by page and allows you to search for a pattern **within the file.** You can search for a pattern by typing "/pattern" where "pattern" is the text you want to search for. For example, the following command displays the contents of the file `example.txt` page by page and allows you to search for the pattern "hello":

```
more example.txt
/hello
```

- **Display a specific number of lines:**

```
more -num file
```

This command displays the specified number of ***lines per page from the file***. The `num` parameter specifies the number of lines per page. For example, the following command displays the contents of the file `example.txt` with 20 lines per page:

```
more -20 example.txt
```

- **Display a file continuously:**

```
tail -f file | more
```

This command displays the contents of the file continuously, as new data is added to the file, and shows the **output page by page**. This can be useful for monitoring log files or other continuously updated files. For example, the following command displays the contents of the file `example.log` continuously and shows the output page by page:

```
tail -f example.log | more
```

- **Display the contents of multiple files:**

```
more file1 file2 file3 ...
```

This command displays the contents of each specified file page by page. You can navigate through the pages by pressing the spacebar to move forward one page, and the "q" key to quit. For example, the following command displays the contents of the files `file1.txt`, `file2.txt`, and `file3.txt` page by page:

```
more file1.txt file2.txt file3.txt
```

less - display the contents of a file page by page [forward and backward navigation]

The `less` command in Linux is a tool that is used to display the contents of a file page by page, similar to the `more` command. However, `less` provides additional features such as backward navigation and searching within the file. Here is an exhaustive explanation of the `less` command with all its variations and most used examples:

- **Basic Syntax:**

```
less file
```

This command displays the contents of the specified file page by page. You can navigate through the pages by pressing the spacebar to move forward one page, and the "b" key to move back one page. You can quit the command by pressing the "q" key. For example, the following command displays the contents of the file `example.txt` page by page:

```
less example.txt
```

▼ Display a file with line numbers:

```
less -N file
```

This command displays the contents of the file ***with line numbers*** and then shows the output page by page. The `-N` option is used to display line numbers. For example, the following command displays the contents of the file `example.txt` with line numbers:

```
less -N example.txt
```

▼ Search for a pattern in a file:

```
less file
/pattern
```

This command displays the contents of the file page by page and allows you to search for a pattern within the file. You can search for a pattern by typing "/pattern" where "pattern" is the text you want to search for. You can navigate through the search results using the "n" key to move forward, and the "N" key to move backward. For example, the following command displays the contents of the file `example.txt` page by page and allows you to search for the pattern "hello":

```
less example.txt
/hello
```

▼ Display a specific number of lines:

```
less -num file
```

This command displays the specified number of lines per page from the file. The `num` parameter specifies the number of lines per page. For example, the following command displays the contents of the file `example.txt` with 20 lines per page:

```
less -20 example.txt
```

▼ Display a file continuously:

```
tail -f file | less
```

This command displays the contents of the file continuously, as new data is added to the file, and shows the output page by page. This can be useful for monitoring log files or other continuously updated files. For example, the following command displays the contents of the file `example.log` continuously and shows the output page by page:

```
tail -f example.log | less
```

Display the contents of multiple files:

```
less file1 file2 file3 ...
```

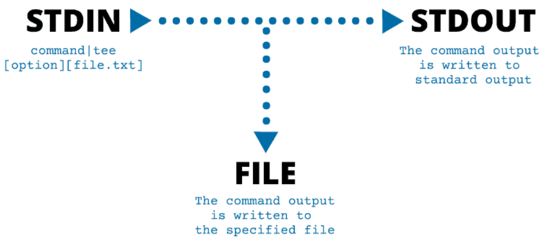
This command displays the contents of each specified file page by page. You can navigate through the pages by pressing the spacebar to move forward one page, and the "b" key to move back one page. You can quit the command by pressing the "q" key. For example, the following command displays the contents of the files `file1.txt`, `file2.txt`, and `file3.txt` page by page:

```
less file1.txt file2.txt file3.txt
```

These are some of the most commonly used variations of the `less` command in Linux. The `less` command is a powerful tool that can help you quickly display the contents of a file page by page, and can be useful in scripting and automation tasks.

tee - Read from standard input and write to standard output and files as well.

The `tee` command in Linux is a tool that is used to ***read from standard input and write to standard output and files simultaneously***. Here is an explanation of the `tee` command with all its variations and most used examples:



Basic Syntax:

```
command | tee file
```

This command reads the ***output of `command`*** from standard input and writes it to both standard output and the specified file simultaneously. For example, the following command writes the output of `ls` command to both standard output and a file named `filelist.txt`:

```
ls | tee filelist.txt
```

Append to a file:

```
command | tee -a file
```

This command appends the output of `command` to the **specified file, rather than overwriting it**. The `-a` option is used to append to the file. For example, the following command appends the output of `ls` command to a file named `filelist.txt`:

```
ls | tee -a filelist.txt
```

Display output on screen and write to file:

```
command | tee file1 file2 ...
```

This command reads the output of `command` from standard input and writes it to both standard output and **multiple specified files simultaneously**. For example, the following command writes the output of `ls` command to both standard output and two files named `filelist1.txt` and `filelist2.txt`:

```
ls | tee filelist1.txt filelist2.txt
```

Display output with color:

```
command --color=always | tee file
```

This command displays the output of `command` with color and writes it to the specified file simultaneously. The `--color=always` option is used to enable color output. For example, the following command writes the color output of `ls` command to a file named `filelist.txt`:

```
ls --color=always | tee filelist.txt
```

Display output with line numbers:

```
command | nl | tee file
```

This command displays the output of `command` with line numbers and writes it to the specified file simultaneously. The `nl` command is used to add line numbers to the output. For example, the following command writes the output of `ls` command with line numbers to a file named `filelist.txt`:

```
ls | nl | tee filelist.txt
```

These are some of the most commonly used variations of the `tee` command in Linux. The `tee` command is a useful tool that can help you write output to both standard output and files simultaneously, and can be useful in scripting and automation tasks.

Stdin, stdou, stderr explained

In Linux, standard input (stdin), standard output (stdout), and standard error (stderr) are three standard communication channels used by programs for input and output. Here's a brief explanation of each:

1. **Standard Input (stdin):**

   This is the default input source of a program. It is typically used for reading data from the keyboard or from a file. When you enter data on the keyboard, it is sent to the program via standard input. Programs can also be designed to read input data from a file instead of the keyboard by redirecting standard input to the file. In Linux, the standard input is represented by the file descriptor `0`.

2. **Standard Output (stdout):**

   This is the default output destination of a program. It is typically used for displaying data on the screen or for writing data to a file. When a program outputs data, it is sent to standard output by default. Programs can also be designed to write output data to a file instead of the screen by redirecting standard output to the file. In Linux, the standard output is represented by the file descriptor `1`.

3. **Standard Error (stderr):**

   This is the default error output destination of a program. It is typically used for displaying error messages on the screen or for writing error messages to a file. When a program encounters an error, it outputs an error message to standard error by default. Programs can also be designed to write error messages to a file instead of the screen by redirecting standard error to the file. In Linux, the standard error is represented by the file descriptor `2`.

Standard input, output, and error can be redirected using special symbols in the shell. Here are some examples:

1. **Redirect standard output to a file:**

```
command > file.txt
```

This command redirects the output of `command` to the file `file.txt`

Suppose you want to list all the files in a directory and save the output to a file named "filelist.txt". You can use the `ls` command to list the files and redirect the output to the file "filelist.txt" using the `>` symbol. Here's the command:

```
ls > filelist.txt
```

This command will execute the `ls` command and redirect its output to the file "filelist.txt". Now, if you open the "filelist.txt" file, you will see a list of all the files in the current directory.

2. **Redirect standard input from a file:**

```
command < file.txt
```

This command redirects the input of `command` from the file `file.txt`.

Suppose you have a program named "myprogram" that reads input from standard input and you want to run the program with input from a file named "input.txt". You can use the `<` symbol to redirect the contents of "input.txt" as input to the "myprogram" program. Here's the command:

```
./myprogram < input.txt
```

This command will execute the "myprogram" program and redirect the contents of "input.txt" as input to the program. Now, the program will read the input from "input.txt" instead of from the keyboard. This can be useful if you have a large amount of input data or if you want to automate the input for testing or other purposes.

3. **Redirect standard error to a file:**

```
command 2> error.txt
```

This command redirects the error output of `command` to the file `error.txt`.

Suppose you have a program named "myprogram" that can produce some error messages and you want to redirect those error messages to a file named "errorlog.txt". You can use the `2>` symbol to redirect error output from the "myprogram" program to the "errorlog.txt" file. Here's the command:

```
./myprogram 2> errorlog.txt
```

4. **Redirect both standard output and standard error to a file:**

```
command > output.txt 2>&1
```

This command redirects both the output and error output of `command` to the file `output.txt`.

if you want to redirect both standard output and standard error to ***separate*** files, you can use the `>` and `2>` symbols together. For example:

```
./myprogram > output.txt 2> errorlog.txt
```

This command will execute the "myprogram" program and redirect standard output to the "output.txt" file and error output to the "errorlog.txt" file.

Standard input, output, and error are important concepts in Linux and are used **extensively in shell scripting and other programming tasks.** Understanding how to redirect these communication channels can be useful for customizing the behavior of programs and for automating tasks.

▼ wc - count lines, words, and characters in a file or standard input

The `wc` command in Linux is a tool that is used to count lines, words, and characters in a file or standard input. Here is an exhaustive explanation of the `wc` command with all its variations and most used examples:

1. **Basic Syntax:**

   ```
   wc file
   ```

   This command displays the number of **lines, words, and characters** in the specified file. For example, the following command displays the number of lines, words, and characters in the file `example.txt`:

   ```
   wc example.txt
   ```

2. **Count lines only:**

   ```
   wc -l file
   ```

   This command displays the number of lines in the specified file. The `-l` option is used to count lines only. For example, the following command displays the number of lines in the file `example.txt`:

   ```
   wc -l example.txt
   ```

3. **Count words only:**

   ```
   wc -w file
   ```

   This command displays the number of words in the specified file. The `-w` option is used to count words only. For example, the following command displays the number of words in the file `example.txt`:

   ```
   wc -w example.txt
   ```

4. **Count characters only:**

   ```
   wc -c file
   ```

   This command displays the number of characters in the specified file. The `-c` option is used to count characters only. For example, the following command displays the number of characters in the file `example.txt`:

   ```
   wc -c example.txt
   ```

5. **Count bytes instead of characters:**

   ```
   wc -m file
   ```

   This command displays the number of bytes in the specified file, rather than characters. The `-m` option is used to count bytes instead of characters. For example, the following command displays the number of bytes in the file `example.txt`:

   ```
   wc -m example.txt
   ```

6. **Count multiple files:**

```
wc file1 file2 file3 ...
```

This command displays the number of lines, words, and characters in each specified file. For example, the following command displays the number of lines, words, and characters in the files `file1.txt`, `file2.txt`, and `file3.txt`:

```
wc file1.txt file2.txt file3.txt
```

These are some of the most commonly used variations of the `wc` command in Linux. The `wc` command is a simple but powerful tool that can help you quickly count lines, words, and characters in a file or standard input, and can be useful in **scripting and automation tasks.**

▼ redirecting and appending (>, >>)

In Linux, the redirect command is used to redirect **input, output, and error streams** from one source to another. Here is an explanation of the redirect command with all its variations and examples:

1. **Redirect standard output to a file:**

```
command > file
```

This command redirects the output of `command` to the specified file **(overwrites the contents)**. For example, the following command redirects the output of the `ls` command to a file named `filelist.txt`:

```
ls > filelist.txt
```

2. **Append standard output to a file:**

```
command >> file
```

This command appends the output of `command` to the specified file. For example, the following command appends the output of the `ls` command to a file named `filelist.txt`:

```
ls >> filelist.txt
```

3. **Redirect standard input from a file:**

```
command < file
```

This command redirects the input of `command` from the specified file. For example, the following command reads the contents of a file named `input.txt` as input to the `wc` command:

```
wc -w < input.txt
```

▼ Pipe (|) - connect output of one command to input of another command

In Linux, the pipe command is used to connect two or more commands in a way that the output of one command is used as input to another command

here's an explanation of the pipe command with all its variations and examples, along with possible output for each example:

1. **Basic Syntax:**

   ```
   command1 | command2
   ```

   This command connects the output of `command1` to the input of `command2`. For example, the following command lists all the files in the current directory and then filters the output to show only the files that contain the word "file":

   ```
   ls | grep file
   ```

   Possible output:

   ```
   file1.txt
   file2.txt
   file3.txt
   ```

2. **Pipe multiple commands:**

   ```
   command1 | command2 | command3 ...
   ```

   This command connects the output of `command1` to the input of `command2`, and then connects the output of `command2` to the input of `command3`, and so on. For example, the following command lists all the files in the current directory, filters the output to show only the files that contain the word "file", and then sorts the output:

   ```
   ls | grep file | sort
   ```

   Possible output:

   ```
   file1.txt
   file2.txt
   file3.txt
   ```

3. **Pipe with awk:**

   ```
   command | awk 'pattern {action}'
   ```

   This command connects the output of `command` to the input of `awk` and provides a way to perform operations on the output using `awk`. The `pattern` and `action` specify the conditions and actions to be performed by `awk`. For example, the following command lists all the files in the current directory and then uses `awk` to print the file name and size:

   ```
   ls -l | awk '{print $9, $5}'
   ```

   Possible output:

   ```
   file1.txt 1024
   file2.txt 2048
   file3.txt 3072
   ```

4. **Pipe with sed:**

```
command | sed 'expression'
```

This command connects the output of `command` to the input of `sed` and provides a way to perform operations on the output using `sed`. The `expression` specifies the operation to be performed by `sed`. For example, the following command lists all the files in the current directory and then uses `sed` to replace the word "file" with "document" in the output:

```
ls | sed 's/file/document/g'
```

Possible output:

```
document1.txt
document2.txt
document3.txt
```

5. **Pipe with xargs:**

```
command1 | xargs command2
```

This command connects the output of `command1` to the input of `xargs`, which is used to pass the output as arguments to `command2`. For example, the following command lists all the files in the current directory and then uses `xargs` to pass the file names as arguments to the `rm` command to delete the files:

```
ls | xargs rm
```

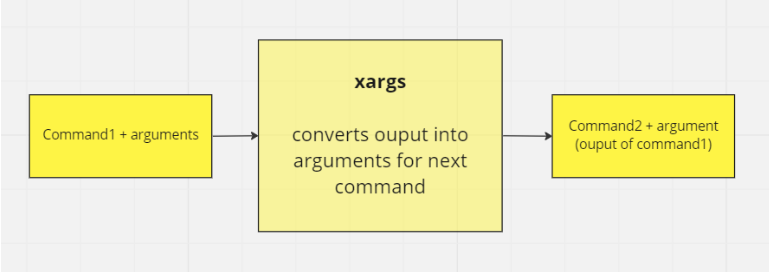
Possible output:

```
rm: cannot remove 'file1.txt': Permission denied
rm: cannot remove 'file2.txt': Permission denied
rm: cannot remove 'file3.txt': Permission denied
```

These are some of the most commonly used variations of the pipe command in Linux. The pipe command is a powerful tool that can be used to connect multiple commands and perform complex operations on the output of one command using another command

xargs - execute arguments

The `xargs` command is used to execute a command from standard input. The name "xargs" is short for "execute arguments". It takes the output from a previous command and passes it as arguments to the command specified. It is commonly used in combination with `find` and `grep` to perform tasks such as finding and deleting files.


Command1 + arguments
xargs
converts ouput into arguments for next command
Command2 + argument (ouput of command1)

here's an explanation of the `xargs` command with all its variations and examples, along with possible output for each example:

1. **Basic Syntax:**

```
command1 | xargs command2
```

This command connects the output of `command1` to the input of `xargs`, which is used to pass the output as arguments to `command2`. For example, the following command lists all the files in the current directory and then uses `xargs` to pass the file names as arguments to the `cat` command to display the contents of the files:

```
ls | xargs cat
```

Possible output:

```
contents of file1.txt
contents of file2.txt
contents of file3.txt
```

2. **Specify the number of arguments:**

```
command | xargs -n num command2
```

This command specifies the ***maximum number of arguments*** to be passed to `command2` at a time. For example, the following command lists all the files in the current directory and then uses `xargs` to pass **two file names *at a time*** to the `cat` command to display the contents of the files:

```
ls | xargs -n 2 cat
```

Possible output:

```
contents of file1.txt
contents of file2.txt

contents of file3.txt
```

3. **Replace placeholders:**

```
command | xargs -I {} command2 {}
```

This command replaces the `{}` placeholder in `command2` with the arguments passed by `xargs`. For example, the following command lists all the files in the current directory and then uses `xargs` to pass the file names as arguments to the `mv` command to move the files to a new directory:

```
ls | xargs -I {} mv {} newdir/
```

This command moves all the files in the current directory to a new directory named `newdir/`.

4. **Ignore empty arguments:**

```
command | xargs -r command2
```

This command ignores any empty arguments passed by `command` to `xargs`. For example, the following command lists all the files in the current directory and then uses `xargs` to pass the file names as arguments to the `rm` command to delete the files, but ignores any empty arguments:

```
ls | grep txt | xargs -r rm
```

Possible output:

```
rm: cannot remove 'file1.txt': Permission denied
rm: cannot remove 'file2.txt': Permission denied
rm: cannot remove 'file3.txt': Permission denied
```

Note that the actual output may vary depending on the contents of the directory and the specific commands used. These are just some examples to demonstrate the possible outputs of the `xargs` command.

▼ difference between pipe command and xargs command explain with example

the difference between the pipe command and the xargs command:

The pipe command (`|`) is like a pipe that connects two commands. It takes the output of one command and sends it as input to another command. For example, if you have a list of files and you want to find the ones that contain a certain word, you can use the pipe command to send the list of files to the `grep` command, which will search for the word in the files and return the results.

The xargs command is like a helper that takes a list of things and uses them as arguments for another command. For example, if you have a list of files and you want to delete them, you can use the xargs command to pass the file names as arguments to the `rm` command, which will delete the files.

In other words, the pipe command connects two commands to transform or filter the output of one command, while the xargs command helps you execute a command with a list of arguments.

There are some commands that are commonly used with the pipe command, but not with the xargs command, and vice versa. Here are some examples:

Commands that are commonly used with pipe command:

1. `grep` : This command is used to search for a pattern in a text file or output. It is commonly used with the pipe command to search for a pattern in the output of another command.
2. `awk` : This command is used to manipulate and process text data. It is commonly used with the pipe command to process the output of another command.
3. `sort` : This command is used to sort lines of text data. It is commonly used with the pipe command to sort the output of another command.

Commands that are commonly used with xargs command:

1. `rm` : This command is used to delete files and directories. It is commonly used with the xargs command to delete multiple files at once.
2. `mkdir` : This command is used to create a new directory. It is commonly used with the xargs command to create multiple directories at once.
3. `find` : This command is used to search for files and directories in a directory hierarchy. It is commonly used with the xargs command to perform actions on the files and directories that are found.

In general, the pipe command is used to connect commands and process the output of one command with another command, while the xargs command is used to build and execute commands from standard input. The choice of command depends on the task at hand and the type of data being processed.

▼ Text Processing

▼ wildcards - Explained

Wildcards in Linux are **special characters** that are used to represent one or more characters in a filename or directory name. They are used with various Linux commands such as ls, cp, mv, rm, and others to perform operations on multiple files at once. Here are some common wildcards and their uses:

1. Asterisk (*) - Represents any number of characters, including zero characters.

   Example: `ls *.txt` - This command will list all the files in the current directory that have a .txt extension.

2. Question mark (?) - Represents a single character.

   Example: `ls file?.txt` - This command will list all the files in the current directory that have the name "file" followed by any single character, followed by the .txt extension.

3. Bracket ([]) - Represents any one of the characters enclosed in the brackets.

   Example: `ls file[123].txt` - This command will list all the files in the current directory that have the name "file" followed by either "1", "2", or "3", followed by the .txt extension.

4. Curly braces ({}) - Represents a set of characters or a range of numbers.

   Example: `cp file{1..5}.txt backup/` - This command will copy files file1.txt, file2.txt, file3.txt, file4.txt, and file5.txt to the backup directory.

Using wildcards in Linux can save time and effort by allowing you to perform operations on multiple files at once. However, it is important to use them carefully and with the correct syntax to avoid unintended consequences.

what are special characters in Linux, which are used as anhors

Special characters in Linux used as anchors are typically used in regular expressions to match the start and end of a line, word or character string. Here are some examples:

1. ^ - The caret character (^) is used as an anchor to match the beginning of a line or string. For example, the regular expression `^hello` will match any line or string that starts with the word "hello".
2. $ - The dollar sign character ($) is used as an anchor to match the end of a line or string. For example, the regular expression `world$` will match any line or string that ends with the word "world".

grep - Search for a pattern WITHin a file.

The `grep` command is a powerful tool used in Unix-based operating systems for searching and filtering text within files or streams. It stands for "global regular expression print".

The basic syntax of the `grep` command is as follows:

```
grep [options] pattern [file...]
```

Here, `pattern` is the regular expression or string that you want to search for, and `[file...]` is the name of the file or files you want to search within. If you don't specify any file name, `grep` reads the input from the standard input (stdin).

Here are some common options used with `grep`:

- `i` or `-ignore-case`: Ignore case distinctions in both the **pattern and the input files**. For example, `grep -i "hello" file.txt` will match "hello", "HELLO", and "Hello".
- `v` or `-invert-match`: Invert the match, i.e., show the lines that do not match the pattern. For example, `grep -v "hello" file.txt` will show all the lines in `file.txt` that do not contain the word "hello".
- `n` or `-line-number`: Prefix each line of output with its line number in the file. For example, `grep -n "hello" file.txt` will show the line numbers of all the lines in `file.txt` that contain the word "hello".
- `r` or `-recursive`: Search for the pattern recursively in all files in the specified directory and its subdirectories. For example, `grep -r "hello" /path/to/directory` will search for the word "hello" in all files in the directory `/path/to/directory` and its subdirectories.
- `w` or `-word-regexp`: Match only whole words. For example, `grep -w "hello" file.txt` will match "hello" but not "hello123" or "helloworld".

  ```
  More Options:
  -c : This prints only a count of the lines that match a pattern
  -h : Display the matched lines, but do not display the filenames.
  -i : Ignores, case for matching
  -l : Displays list of a filenames only.
  -n : Display the matched lines and their line numbers.
  -v : This prints out all the lines that do not matches the pattern
  -e exp : Specifies expression with this option. Can use multiple times.
  -f file : Takes patterns from file, one per line.
  -E : Treats pattern as an extended regular expression (ERE)
  -w : Match whole word
  -o : Print only the matched parts of a matching line,
   with each such part on a separate output line.
  ```

```
-A n: Prints searched line and nlines after the result.
-B n : Prints searched line and n line before the result.
-C n : Prints searched line and n lines after before the result.
```

Here are some examples of how to use the `grep` command:

1. **Searching for a word in a file:**

   ```
   grep "hello" file.txt
   ```

   This command will search for the word "hello" in the file `file.txt` and display all the lines that contain the word "hello".

2. **Searching for a word in multiple files:**

   ```
   grep "hello" file1.txt file2.txt file3.txt
   ```

   This command will search for the word "hello" in the files `file1.txt`, `file2.txt`, and `file3.txt` and display all the lines that contain the word "hello".

3. **Searching for a word in a directory:**

   ```
   grep "hello" /path/to/directory/*.txt
   ```

   This command will search for the word "hello" in all the `.txt` files in the directory `/path/to/directory` and display all the lines that contain the word "hello".

4. **Searching for a word recursively in a directory:**

   ```
   grep -r "hello" /path/to/directory/
   ```

   This command will search for the word "hello" in all files in the directory `/path/to/directory` and its subdirectories and display all the lines that contain the word "hello".

5. **Searching for a word ignoring case:**

   ```
   grep -i "hello" file.txt
   ```

   This command will search for the word "hello" in the file `file.txt` ignoring case and display all the lines that contain "hello", "Hello", "HELLO", etc.

6. **Searching for lines that do not contain a word:**

   ```
   grep -v "hello" file.txt
   ```

   This command will search for all the lines in the file `file.txt` that do not contain the word "hello" and display them.

7. **Searching for whole words only:**

   ```
   grep -w "hello" file.txt
   ```

   This command will search for the whole word "hello" in the file `file.txt` and display all the lines that contain the word "hello" as a separate word.

8. **Searching for a regular expression:**

   ```
   grep "^[0-9]" file.txt
   ```

   This command will search for all the lines in the file `file.txt` that start with a number.

The output of the `grep` command depends on the options used and the content of the files being searched. In general, the output consists of the lines that match the pattern, with the matching text highlighted or displayed in some way. When using the `-n` option, the line numbers are also displayed. If no matching lines are found, `grep` produces no output.

▼ cut - Remove sections from each line of a file.

The `cut` command is a Unix-based command-line tool used to extract sections of text from a file or input stream. It is commonly used to extract specific columns from a delimited file, such as a CSV or tab-separated file.

The basic syntax of the `cut` command is as follows:

```
cut [options] [file...]
```

Here, `[file...]` is the name of the file or files you want to extract data from. If no files are specified, `cut` reads from the standard input (stdin).

Here are some common options used with `cut`:

- `c` or `-characters`: Extract specified characters from each line. For example, `cut -c 1-5 file.txt` will extract the first five characters of each line in `file.txt`.
- `f` or `-fields`: Extract specified fields (columns) from each line. For example, `cut -f 2,4,6 file.txt` will extract the second, fourth, and sixth fields from each line in `file.txt`.
- `d` or `-delimiter`: Set the delimiter used to separate fields. For example, `cut -d "," -f 1,3 file.csv` will extract the first and third fields from each line in the CSV file `file.csv`, assuming that the fields are separated by commas.
- `s` or `-only-delimited`: Only output lines that contain the delimiter. For example, `cut -d "," -f 1,3 -s file.csv` will only output lines that contain a comma, and extract the first and third fields from those lines.

here's an exhaustive explanation of the `cut` command with all its variations and examples, along with possible output for each example:

1. **Basic Syntax:**

   ```
   cut -f fields file
   ```

   This command extracts specific columns (`fields`) from a file. By default, fields are separated by a tab.

   For example, the following command extracts the first column of a file named `data.txt`:

   ```
   cut -f 1 data.txt
   ```

   Possible output:

   ```
   John
   Sara
   Tom
   ```

2. **Change field separator:**

   ```
   cut -d delimiter -f fields file
   ```

   This command extracts specific **columns (**`fields`) from a file, using a specified delimiter (`delimiter`) to separate fields.

   For example, the following command extracts the first column of a file named `data.csv`, which uses a comma as a delimiter:

   ```
   cut -d ',' -f 1 data.csv
   ```

Possible output:

```
John
Sara
Tom
```

3. **Select a range of fields:**

```
cut -f start-end file
```

This command extracts a range of columns (`start` to `end`) from a file.

For example, the following command extracts the second and third columns of a file named `data.txt`:

```
cut -f 2-3 data.txt
```

Possible output:

```
25  New York
30  Los Angeles
40  Chicago
```

4. **Combine multiple fields:**

```
cut -f field1,field2 file
```

This command extracts multiple columns (`field1` and `field2`)

from a file. For example, the following command extracts the first and third columns of a file named `data.txt`:

```
cut -f 1,3 data.txt
```

Possible output:

```
John  New York
Sara  Los Angeles
Tom   Chicago
```

5. Output delimiter:

```
cut -d delimiter output_delimiter delimiter -f fields file
```

This command extracts specific columns (`fields`) from a file, using a specified delimiter (`delimiter`) to separate fields, and a specified delimiter (`output_delimiter`) to separate the output fields. For example, the following command extracts the first and third columns of a file named `data.csv`, which uses a comma as a delimiter, and separates the output fields with a colon:

```
cut -d ',' --output-delimiter ':' -f 1,3 data.csv
```

Possible output:

```
John:25
Sara:30
Tom:40
```

6. **Cut by character position:**

   ```
   cut -c characters file
   ```

   This command extracts specific characters (`characters`) from a file.

   For example, the following command extracts the first **five characters of each line** from a file named `data.txt` :

   ```
   cut -c 1-5 data.txt
   ```

   Possible output:

   ```
   John
   Sara
   Tom
   ```

   These are just a few examples of the many variations of the `cut` command. The actual output may vary depending on the contents of the file and the specific commands used.

▼ sed - Stream editor.

The `sed` command is a Unix-based command-line tool used for **manipulating** text. It stands for "stream editor" **[not an conventional text editor]**. `sed` can be used to perform a wide range of text transformations, including **search and replace, insertion and deletion of lines**, and more advanced operations using regular expressions.

The basic syntax of the `sed` command is as follows:

```
sed [options] [script] [file...]
```

Here, `[script]` is a series of `sed` commands that specify the text transformations to be performed, and `[file...]` is the name of the file or files to be edited. If no files are specified, `sed` reads from the standard input (stdin).

Here are some common options used with `sed` :

- `i` or `-in-place` : Edit files **in place, i.e., modify** the files directly rather than writing to standard output.
- `e` or `-expression` : Add a script of `sed` commands to the command line.
- `n` or `-quiet` : Suppress automatic printing of pattern space.
- `r` or `-regexp-extended` : Use extended regular expressions.
- `s` or `-separate` : Separate multiple commands with semicolons.

here's an exhaustive explanation of the `sed` command with all its variations and examples, along with possible output for each example:

1. **Basic Syntax:**

   ```
   sed 's/pattern/replacement/g' file
   ```

   This command searches for a `pattern` in a file and replaces it with a `replacement` string. The `g` at the end of the command means that the replacement should be done **globally (on all occurrences of the pattern).**

   For example, the following command replaces all occurrences of the word "hello" with "hi" in a file named `data.txt` :

   ```
   sed 's/hello/hi/g' data.txt
   ```

Possible output:

```
Hi, how are you?
I'm doing fine, hi!
```

2. **Replace in-place:**

```
sed -i 's/pattern/replacement/g' file
```

This command performs the same function as the basic syntax above, but the changes are made in-place, that is, the original file is modified.

For example, the following command replaces all occurrences of the word "hello" with "hi" in a file named `data.txt`, and saves the changes to the file:

```
sed -i 's/hello/hi/g' data.txt
```

3. **Use regular expressions:**

```
sed 's/regex/replacement/g' file
```

This command uses regular expressions to search for a pattern in a file and replace it with a replacement string. **Regular expressions** are a powerful way of matching patterns in text data.

For example, the following command replaces all occurrences of a number with the word "number" in a file named `data.txt`:

```
sed 's/[0-9]+/number/g' data.txt
```

Possible output:

```
Hello, how are you?
I have a number of things to do today.
```

4. **Addressing specific lines:**

```
sed '2s/pattern/replacement/g' file
```

This command searches for a pattern in a specific line (in this case, line 2) of a file and replaces it with a replacement string. For example, the following command replaces the word "hello" with "hi" in line 2 of a file named `data.txt`:

```
sed '2s/hello/hi/g' data.txt
```

5. **Print specific lines:**

```
sed -n '2p' file
```

This command prints a specific line (in this case, line 2) of a file. The `-n` option means that the command should not print every line of the file, and the `p` command tells `sed` to print the line. For example, the following command prints line 2 of a file named `data.txt`:

```
sed -n '2p' data.txt
```

Possible output:

```
I'm doing fine, hello!
```

6. **Combining multiple commands:**

```
sed 's/hello/hi/g; 2p' file
```

This command combines multiple `sed` commands using a semicolon. In this case, the command replaces all occurrences of the word "hello" with "hi" and then prints line 2 of the file. For example, the following command replaces all occurrences of the word "hello" with "hi" and then prints line 2 of a file named `data.txt`:

```
sed 's/hello/hi/g; 2p' data.txt
```

Possible output:

```
Hi, how are you?
I'm doing fine, hi!
```

These are just a few examples of the many variations of the `sed` command. The actual output may vary depending on the contents of the file and the specific commands used.

7. **Deleting lines from a particular file :** SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the fileExamples:1. To Delete a particular line say n in this example

```
Syntax:
$ sed 'nd' filename.txt
```

2. To Delete a last line

```
Syntax:
$ sed '$d' filename.txt
```

3. To Delete line from range x to y

```
Example:
$ sed '3,6d' filename.txt
```

4. To Delete from nth to last line

```
Example:
$ sed '12,$d' filename.txt
```

5. To Delete pattern matching line

```
Example:
$ sed '/abc/d' filename.txt
```

8. **Appending text to a file:**

```
sed '$a\new_text' file.txt
```

This command will append the text `new_text` to the end of the file `file.txt`.

▼ awk - Pattern scanning and processing language.

Awk is a powerful command-line tool that is used for **text processing and data analysis.** It is particularly useful for working with **structured data,** such as data in **CSV files or log files**. The name awk comes from the initials of its three creators - Alfred Aho , Peter Weinberger, and Brian Kernighan. These three computer scientists at Bell Labs developed awk in the late 1970s as a tool for text processing and data analysis. Awk works by reading input files line by line, splitting each line **into fields**, and then performing actions on those fields based on patterns and rules specified by the user. Here are some of the most common variations of the awk command, along with examples and possible output:

Here are some variations of the awk command and their examples with possible output:

1. **Printing Specific Column(s):**

```
awk '{print $1, $3}' filename
```

This command prints the first and third columns of the file 'filename'.

2. **Using a Delimiter:**

```
awk -F: '{print $1, $3}' filename
```

This command prints the first and third columns of the file 'filename', using a colon (:) as the delimiter.

3. **Using Conditional Statements:**

```
awk '{if ($3 > 1000) print $1}' filename
```

This command prints the first column of the file 'filename' if the value in the third column is greater than 1000.

4. **Using Built-in Variables:**

```
awk '{print NR, $1}' filename
```

This command prints the line number (NR) and the first column of the file 'filename'.

5. **Using Regular Expressions:**

```
awk '/pattern/ {print $1}' filename
```

This command searches for the pattern in the file 'filename' and prints the first column of the matching lines.

6. **Using Multiple Commands:**

```
awk '{print $1; print $2}' filename
```

This command prints the first and second columns of the file 'filename' **on separate lines.**

7. **Using Script File:**

```
awk -f scriptfile filename
```

This command executes the awk commands stored in the script file 'scriptfile' on the file 'filename'.

8. **Redirecting Output:**

```
awk '{print $1}' filename > outputfile
```

This command redirects the output of the awk command to a file named 'outputfile'.

Example Output: Suppose we have a file named 'data.txt' with the following content:

```
John 35 2000
Mary 27 1500
Tom 40 3000
```

Running the command `awk '{print $1, $3}' data.txt` would produce the following output:

```
John 2000
Mary 1500
Tom 3000
```

Running the command `awk '{if ($3 > 1000) print $1}' data.txt` would produce the following output:

```
John
Mary
Tom
```

Running the command `awk '/Tom/ {print $2}' data.txt` would produce the following output:

```
40
```

These are just a few examples of the awk command and its variations. There are many more features and options that can be used to perform complex text processing tasks.

How to print all the contents of the file using `awk`

To print *all* the contents of a file, the action you specify inside the curly braces is `print $0`.

This will work in exactly the same way as the `cat` command mentioned previously.

```
awk '{print $0}' information.txt
```

Ouptut:

```
fristName       lastName        age     city       ID

Thomas          Shelby          30      Rio        400
Omega           Night           45      Ontario    600
Wood            Tinker          54      Lisbon     N/A
Giorgos         Georgiou        35      London     300
Timmy           Turner          32      Berlin     N/A
```

If you would like each line to have a line-number count, you would use the `NR` built-in variable:

```
awk '{print NR,$0}' information.txt
```

```
1 fristName     lastName        age     city       ID
2
3 Thomas        Shelby          30      Rio        400
4 Omega         Night           45      Ontario    600
```

```
5 Wood          Tinker          54      Lisbon     N/A
6 Giorgos       Georgiou        35      London     300
7 Timmy         Turner          32      Berlin     N/A
```

How to print specific columns using `awk`

When using `awk`, you can specify certain columns you want printed.

To have the first column printed, you use the command:

```
awk '{print $1}' information.txt
```

Ouput:

```
Thomas
Omega
Wood
Giorgos
Timmy
```

The `$1` stands for the first field, in this case the first column.

To print the second column,you would use `$2`:

```
awk '{print $2}' information.txt
```

Output:

```
lastName

Shelby
Night
Tinker
Georgiou
Turner
```

The way `awk` determines where each column starts and **ends is with a space, by default.**

To print more than one column, for example the first and forth columns, you would do:

```
awk '{print $1, $4}' information.txt
```

Ouput:

```
fristName city

Thomas    Rio
Omega     Ontario
Wood      Lisbon
Giorgos   London
Timmy     Berlin
```

The `$1` represents the first input field (first column), and the `$4` represents the forth. You separate them with a comma, `$1,$4`, so the output has a space and is more readable.

To print the last field (the last column), you can also use `$NF` which represents the ***last* field in a record:**

```
awk '{print $NF}' information.txt
```

Output:

```
ID
```

```
400
600
N/A
300
N/A
```

To print pen ultimate column, using NF option

```
awk '{print $(NF-1)}' column.txt
```

How to print out lines with a specific pattern in `awk`

You can print a line that **starts** with a specific letter.

For example:

```
awk '/^O/' information.txt
```

Output:

```
Omega           Night           45      Ontario    600
```

That command selects any line with text that *starts* with an `O`.

You use the up arrow symbol (`^`) first, which indicates the beginning of a line, and then the letter you want a line to start with.

You can also print a line that **ends** in a specific pattern:

```
awk '/0$/' information.txt
```

Output:

```
Thomas          Shelby          30      Rio        400
Omega           Night           45      Ontario    600
Giorgos         Georgiou        35      London     300
```

This prints out the lines that end in a `0` – the `$` symbol is used after a character to siginify how a line will end.

That command could also be changed to:

```
awk '! /0$/' information.txt
```

The `!` is used as a `NOT`, so in this case it selects the lines that DON'T end in a `0`.

```
fristName       lastName        age     city       ID

Wood            Tinker          54      Lisbon     N/A
Timmy           Turner          32      Berlin     N/A
```

How to use regular expressions in `awk`

To output words that contain certain letters and print out words that match a pattern you specify, you again use the slashes, `//`, shown previously.

If you want to look for words containing `on`, you'd do:

```
awk ' /io/{print $0}' information.txt
```

Output:

```
Thomas          Shelby          30      Rio        400
Omega           Night           45      Ontario    600
Giorgos         Georgiou        35      London     300
```

This matches all entries that contain `io`.

Say you had an extra column – a `department` column:

```
fristName       lastName        age     city       ID   department

Thomas          Shelby          30      Rio        400  IT
Omega           Night           45      Ontario    600  Design
Wood            Tinker          54      Lisbon     N/A  IT
Giorgos         Georgiou        35      London     300  Data
Timmy           Turner          32      Berlin     N/A  Engineering
```

To find all the information of people working in `IT`, you would need to speficy the string you're searching for between the slashes, `//`:

```
awk '/IT/' information.txt
```

Output:

```
Thomas          Shelby          30      Rio        400  IT
Wood            Tinker          54      Lisbon     N/A  IT
```

What if you wanted to see only the first and last names of the people working in `IT`?

You can specify the column like such:

```
awk '/IT/{print $1, $2}' information.txt
```

Output:

```
Thomas Shelby
Wood   Tinker
```

This will only display the first and second columns where `IT` appears, instead of presenting all fields.

When searching for words with a specific pattern, there may be times when you'll need to use an escape character, like such:

```
awk '/N\/A$/' information.txt
```

Output:

```
Wood            Tinker          54      Lisbon     N/A
Timmy           Turner          32      Berlin     N/A
```

I wanted to find lines that end with the pattern `N/A`.

So, when searching between the `' // '` like shown so far, I had to use an escape character (`\`) between `N/A`, otherwise I would've gotten an error.

How to use comparisson operators in `awk`

If, for example, you wanted to find all the information of employees that were under the age of `40`, you would use the `<` comparisson operator like so:

```
awk '$3 <  40 { print $0 }' information.txt
```

Output:

```
Thomas          Shelby          30      Rio       400
Giorgos         Georgiou        35      London    300
Timmy           Turner          32      Berlin    N/A
```

The output shows only the information of people under 40.

▼ sort - Sort lines of text files.**

The sort command is a Unix/Linux command used to **sort the contents of files or standard input** **alphabetically or numerically**. It can also be used to merge and sort multiple files. The basic syntax of the sort command is as follows:

```
sort [option] [file]
```

Here are some variations of the sort command and their examples with possible output:

1. **Sort a File:**

   ```
   sort filename
   ```

   This command sorts the contents of the file 'filename' in **alphabetical order.**

2. **Reverse Sort:**

   ```
   sort -r filename
   ```

   This command sorts the contents of the file 'filename' in reverse alphabetical order.

3. **Numeric Sort:**

   ```
   sort -n filename
   ```

   This command sorts the contents of the file 'filename' in numerical order.

4. **Case-insensitive Sort:**

   ```
   sort -f filename
   ```

   This command sorts the contents of the file 'filename' in alphabetical order, **ignoring the case of the letters.**

5. **Sort by Column:**

   ```
   sort -k n filename
   ```

   This command sorts the contents of the file 'filename' based on the n-th column.

6. **Merge and Sort Multiple Files:**

   ```
   sort file1 file2 file3
   ```

   This command merges and sorts the contents of the files 'file1', 'file2', and 'file3' in alphabetical order.

7. **Specify Delimiter:**

   ```
   sort -t: -k2 filename
   ```

This command sorts the contents of the file 'filename' based on the 2nd column, using a colon (:) as the delimiter.

8. **Ignore Leading Characters:**

   ```
   sort -k2.2 filename
   ```

   This command sorts the contents of the file 'filename' based on the 2nd column, **ignoring** the **first two characters of the column.**

Example Output: Suppose we have a file named 'data.txt' with the following content:

```
apple 50
banana 30
cherry 20
```

Running the command `sort data.txt` would produce the following output:

```
apple 50
banana 30
cherry 20
```

Running the command `sort -r data.txt` would produce the following output:

```
cherry 20
banana 30
apple 50
```

Running the command `sort -n data.txt` would produce the following output:

```
cherry 20
banana 30
apple 50
```

Running the command `sort -k 2 data.txt` would produce the following output:

```
cherry 20
banana 30
apple 50
```

Running the command `sort -t: -k2 data.txt` would produce an error message, as there is no delimiter in the file.

These are just a few examples of the sort command and its variations. There are many more features and options that can be used to perform complex sorting tasks.

▼ uniq - Report or omit repeated lines.**

The uniq command is a Unix/Linux command used to remove or display duplicate lines from a file or standard input. The basic syntax of the uniq command is as follows:

```
uniq [option] [file]
```

Here are some variations of the uniq command and their examples with possible output:

1. **Remove Duplicate Lines:**

   ```
   uniq filename
   ```

   This command removes duplicate lines from the file 'filename' and displays the remaining lines.

2. **Count Duplicate Lines:**

```
uniq -c filename
```

This command counts the number of occurrences of each line in the file 'filename' and displays the count along with the line.

3. **Ignore Case:**

```
uniq -i filename
```

This command ignores the case of the letters when comparing lines in the file 'filename'.

4. **Only Display Duplicate Lines:**

```
uniq -d filename
```

This command displays only the lines that are duplicated in the file 'filename'.

5. **Specify Delimiter:**

```
uniq -f n -d filename
```

This command displays only the lines that are duplicated in the file 'filename', ignoring the first n fields.

6. **Show Non-duplicate Lines:**

```
uniq -u filename
```

This command displays only the lines that are not duplicated in the file 'filename'.

Example Output: Suppose we have a file named 'data.txt' with the following content:

```
apple
banana
cherry
banana
cherry
```

Running the command `uniq data.txt` would produce the following output:

```
apple
banana
cherry
```

Running the command `uniq -c data.txt` would produce the following output:

```
      1 apple
      2 banana
      2 cherry
```

Running the command `uniq -i data.txt` would produce the same output as `uniq data.txt`.

Running the command `uniq -d data.txt` would produce the following output:

```
banana
cherry
```

Running the command `uniq -f 1 -d data.txt` would produce the following output:

```
cherry
```

Running the command `uniq -u data.txt` would produce the following output:

```
apple
```

These are just a few examples of the uniq command and its variations. There are many more features and options that can be used to perform complex tasks related to duplicate lines in a file.

▼ diff - Compare two files.**

The diff command is a Unix/Linux command used to **compare and display the differences between two files or directories**.

> 💡 **Useful to check the difference between the latest backup file vs the recent backup file**

The basic syntax of the diff command is as follows:

```
diff [option] file1 file2
```

Here are some variations of the diff command and their examples with possible output:

1. **Compare Two Files:**

   ```
   diff file1 file2
   ```

   This command compares the contents of 'file1' and 'file2' and displays the differences between them.

2. **Ignore Case:**

   ```
   diff -i file1 file2
   ```

   This command compares the contents of 'file1' and 'file2' ignoring the case of the letters and displays the differences between them.

3. **Ignore Whitespace:**

   ```
   diff -w file1 file2
   ```

   This command compares the contents of 'file1' and 'file2', ignoring all whitespace characters and displays the differences between them.

4. **Side-by-Side Comparison:**

   ```
   diff -y file1 file2
   ```

   This command performs a side-by-side comparison of 'file1' and 'file2' and displays the differences between them.

5. **Show Context:**

   ```
   diff -c file1 file2
   ```

   This command displays the differences between 'file1' and 'file2', with context lines to show where the differences occur.

6. **Show Unified Context:**

   ```
   diff -u file1 file2
   ```

   This command displays the differences between 'file1' and 'file2', with unified context format.

7. **Ignore Blank Lines:**

   ```
   diff -B file1 file2
   ```

   This command compares the contents of 'file1' and 'file2', ignoring all blank lines and displays the differences between them.

8. **Recursive Comparison:**

   ```
   diff -r dir1 dir2
   ```

   This command recursively compares the contents of 'dir1' and 'dir2', including all subdirectories, and displays the differences between them.

Example Output: Suppose we have two files named 'file1.txt' and 'file2.txt' with the following content:

file1.txt:

```
apple
banana
cherry
```

file2.txt:

```
apple
banana
date
```

Running the command `diff file1.txt file2.txt` would produce the following output:

```
3c3
< cherry
---
> date
```

Running the command `diff -i file1.txt file2.txt` would produce the same output as the previous command.

Running the command `diff -w file1.txt file2.txt` would produce the same output as the previous command.

Running the command `diff -y file1.txt file2.txt` would produce the following output:

```
apple                            apple
banana                           banana
cherry                           date
```

Running the command `diff -c file1.txt file2.txt` would produce the following output:

```
*** file1.txt 2023-03-28 13:30:00.000000000 +0000
--- file2.txt 2023-03-28 13:30:00.000000000 +0000
***************
*** 1,3 ****
  apple
  banana
! cherry
--- 1,3 ----
  apple
  banana
! date
```

Running the command `diff -u file1.txt file2.txt` would produce the following output:

```
--- file1.txt 2023-03-28 13:30:00.000000000 +0000
+++ file2.txt 2023-03-28 13:30:00.000000000 +0000
@@ -1,3 +1,3 @@
 apple
 banana
```

```
-cherry
+date
```

These are just a few examples of the diff command and its variations. There are many more features and options that can be used to perform complex file and directory comparisons.

▼ nl - Number lines of a file.**

The nl command is a Linux/Unix command used to add line numbers to a file or standard input. The basic syntax of the nl command is as follows:

```
nl [option] filename
```

Here are some common options of the nl command:

1. b: Specifies the line numbering mode. The argument can be a, t or n. 'a' numbers all lines, 't' numbers only non-empty lines and 'n' does not add any line numbers.
2. i: Specifies the line number increment. The argument is an integer value.
3. v: Specifies the line number format. The argument can be a style for formatting line numbers, such as '1', '01', '001', etc.
4. w: Specifies the width of the line number field. The argument is an integer value.

Example Output:Suppose we have a file named 'data.txt' with the following content:

```
apple
banana
cherry
```

Running the command `nl data.txt` would produce the following output:

```
1  apple
2  banana
3  cherry
```

Running the command `nl -b a data.txt` would produce the same output as the previous command.

Running the command `nl -b t data.txt` would produce the following output:

```
1  apple
2  banana
3  cherry
```

Running the command `nl -i 2 data.txt` would produce the following output:

```
1  apple
3  banana
5  cherry
```

Running the command `nl -v 01 data.txt` would produce the following output:

```
01 apple
02 banana
03 cherry
```

Running the command `nl -w 4 data.txt` would produce the following output:

```
   1 apple
   2 banana
   3 cherry
```

These are just a few examples of the nl command and its options. There are many more features and options that can be used to customize line numbering in a file or standard input.

▼ Process Management

▼ top - Display system resource usage and processes - table of processes

The top command is a Linux/Unix command used to monitor system processes and their resource usage in real-time. It provides a dynamic, interactive view of system processes, CPU usage, memory usage, and other system metrics. The name "top" is a short form of "**table of processes**" or "top of processes", which describes the command's primary function of displaying a table of running processes on a system with the most resource-intensive processes listed at the top.. The basic syntax of the top command is as follows:

```
top [option]
```

Here are some common options of the top command:

1. d: Specifies the update interval for displaying system metrics. The argument is in seconds.
2. n: Specifies the number of iterations to display before exiting.
3. p: Specifies the process ID to monitor.
4. u: Specifies the user name to monitor.
5. H: Displays individual threads of a process.

Example Output:Running the top command without any options would display the following output:

```
top - 16:18:48 up  1:12,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  79 total,   1 running,  78 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3867.6 total,    864.7 free,   1844.4 used,   1158.4 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   1759.7 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  593 root      20   0  238864  61256  37716 S   0.3   1.6   0:13.15 Xorg
  845 root      20   0  626872  98868  28072 S   0.3   2.5   0:05.68 gnome-shell
  910 root      20   0  365096  19716  10908 S   0.3   0.5   0:01.95 NetworkManager
  955 root      20   0  166812   5288   3856 S   0.3   0.1   0:00.55 systemd-journald
  968 root      20   0  165476   4276   2928 S   0.3   0.1   0:00.38 systemd-logind
 1099 root      20   0  110120   3888   3148 R   0.3   0.1   0:00.05 top
    1 root      20   0  166808  10024   6916 S   0.0   0.3   0:02.80 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.04 kthreadd
    3 root      20   0       0      0      0 S   0.0   0.0   0:00.04 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0   0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kworker/u256:0+
    7 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_sched
    8 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0.0   0.0   0:00.02 migration/0
   10 root      rt   0       0      0      0 S   0.0
```

Sure, here is an explanation of the parameters in the `top` command's output:

- `top`: the name of the command being executed.
- `16:18:48`: the current time.
- `up 1:12`: the system uptime.
- `1 user`: the number of logged-in users.
- `load average: 0.00, 0.00, 0.00`: the system load average for the last 1, 5, and 15 minutes. In this case, the load average is very low, indicating that the system is not under heavy load.
- `Tasks`: the total number of tasks (processes/threads) running on the system.
- `79 total`: the total number of tasks.
- `1 running`: the number of tasks currently running.
- `78 sleeping`: the number of tasks currently sleeping (idle).
- `0 stopped`: the number of tasks stopped by a signal.
- `0 zombie`: the number of zombie tasks (terminated but not yet cleaned up).

- `%Cpu(s)`: the CPU usage breakdown. `us` is the percentage of CPU time spent in user space, `sy` is the percentage spent in system (kernel) space, `ni` is the percentage spent on low priority tasks (nice value), `id` is the percentage of idle CPU time, **`wa` is the percentage of time spent waiting for IO**, `hi` is the percentage of CPU time spent handling hardware interrupts, `si` is the percentage of CPU time spent handling software interrupts, and `st` is the percentage of stolen CPU time (in a virtualized environment).
- `MiB Mem`: the memory usage breakdown. `total` is the total amount of memory, `free` is the amount of free memory, `used` is the amount of used memory, and `buff/cache` is the amount of memory used for buffering and caching.
- `MiB Swap`: the swap usage breakdown. `total` is the total amount of swap space, `free` is the amount of free swap space, `used` is the amount of used swap space, and `avail Mem` is the amount of available memory (including swap space).
- `PID`: the process ID of the task.
- `USER`: the username of the user running the task.
- `PR`: the priority of the task.
- `NI`: the nice value of the task.
    - **How to change the nice value of a task?**
- `VIRT`: the total amount of virtual memory used by the task.
- `RES`: the amount of resident memory used by the task.
- `SHR`: the amount of shared memory used by the task.
- `S`: the task state (running, sleeping, etc.).
- `%CPU`: the percentage of CPU time used by the task.
- `%MEM`: the percentage of memory used by the task.
- `TIME+`: the total CPU time used by the task.
- `COMMAND`: the command name or full path of the task.

ps - Display process status. [less of top]

The ps command is a Linux/Unix command used to display information about **running processes on a system**. It provides a snapshot of the current system state, including process IDs, CPU usage, memory usage, and other system metrics. The basic syntax of the ps command is as follows:

```
ps [option]
ps -ef
```

Here are some common options of the ps command:

1. e: Displays information about all running processes.
2. f: Displays a full-format listing of process information, including user, CPU usage, and command-line arguments.
3. u: Displays process information for a specific user.
4. p: Displays process information for a specific process ID.
5. C: Displays process information for a specific command.

Example Output:Running the `ps` command without any options would display the following output:

```
PID TTY          TIME CMD
1 ?        00:00:01 systemd
2 ?        00:00:00 kthreadd
3 ?        00:00:00 ksoftirqd/0
5 ?        00:00:00 kworker/0:0H
7 ?        00:00:00 rcu_sched
8 ?        00:00:00 rcu_bh
9 ?        00:00:00 migration/0
10 ?        00:00:00 watchdog/0
11 ?        00:00:00 watchdog/1
12 ?        00:00:00 migration/1
13 ?        00:00:00 ksoftirqd/1
```

```
girish@girish-Inspiron-5570:~/Downloads/umask$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1 168484 11460 ?        Ss   05:33   0:06 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    05:33   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   05:33   0:00 [rcu_gp]
```

▼ **difference between ps and top command**

The ps command displays a list of the currently running processes on a Linux or Unix-based system, including the process ID, CPU utilization, memory usage, and more.

The top command also displays a list of currently running processes, but it **provides more detailed information** than the ps command and updates the information in real-time, so it can be used to monitor system performance and identify processes that are consuming system resources. top also updates the information in real-time, so it can be used to monitor system performance and identify processes that are consuming system resources.

▼ kill - Send a signal to a process to terminate it.

The kill command is a Linux/Unix command used to **terminate or signal processes on a system**. It sends a signal to a specified process or group of processes, prompting them to perform a specific action, such as **terminating, reloading, or stopping.**

The basic syntax of the kill command is as follows:

```
kill [signal] [process ID(s)]
```

Here are some common signals that can be used with the kill command:

1. SIGTERM (`15`): This signal prompts the process to terminate gracefully, allowing it to perform any necessary cleanup tasks before exiting.
2. SIGKILL (`9`): This signal forces the process to terminate immediately, without allowing it to perform any cleanup tasks.
3. SIGHUP (`1`): This signal prompts the process to reload its configuration or restart.

   Example Output: Here are some examples of using the kill command with different options and signals:

   1. **To terminate a process with a specific process ID: (by default - 15)**

      ```
      kill 1234
      ```

      This command sends a SIGTERM signal to the process with the ID of 1234, prompting it to terminate **gracefully**.

   2. **To force a process to terminate immediately:**

      ```
      kill -9 1234
      ```

      This command sends a SIGKILL signal to the process with the ID of 1234, forcing it to terminate immediately without performing any cleanup tasks.

   3. **To terminate multiple processes at once:**

      ```
      kill 1234 5678 91011
      ```

      This command sends a SIGTERM signal to the processes with the IDs of 1234, 5678, and 91011, prompting them to terminate **gracefully**.

   In summary, the kill command is a powerful tool for managing processes on a Linux/Unix system. It allows you to terminate, signal, and control processes, providing a flexible and efficient way to manage system resources. However, it should be used with caution, as sending the wrong signal or terminating the wrong process can have serious consequences for system stability and data integrity.

▼ iostat - Monitor system input/output (I/O) statistics for devices and partitions.

The `iostat` command is a Linux/Unix utility that is used to monitor system input/output (I/O) statistics for devices and partitions. It displays statistics about CPU utilization, I/O operations per second (IOPS), disk throughput, and other I/O-related metrics.

The `iostat` command can be used to monitor the performance of storage devices, such as hard drives and solid-state drives (SSDs), and to identify performance bottlenecks or issues with disk I/O.

The `iostat` command has several options that allow you to customize the output, such as:

- `c` or `-cpu`: display CPU utilization statistics.
- `d` or `-device`: display statistics for each device separately.
- `k` or `-kilobytes`: display statistics in kilobytes per second.
- `m` or `-megabytes`: display statistics in megabytes per second.
- `N` or `-noheaders`: do not display column headers.
- `t` or `-timestamp`: display a timestamp for each report.

Here is an example of `iostat` command output:

```
Linux 5.4.0-77-generic (hostname)  05/03/2023  _x86_64_    (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.75    0.00    0.51    0.09    0.00   96.65

Device            tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd
sda               0.95        11.62         0.06         0.00   12590648     719350          0
sdb               0.02         0.07         0.00         0.00      75268          0          0
```

In this example, the `avg-cpu` section displays the CPU utilization statistics, and the `Device` section displays the I/O statistics for each device (`sda` and `sdb`). The `tps` column shows the number of I/O operations per second, while the `kB_read/s` and `kB_wrtn/s` columns show the amount of data read and written, respectively, in kilobytes per second.

Troubleshooting IO performance issues on Linux

In this video we explain how to diagnose and troubleshoot step by step what is causing disk io performance issues on linux servers.

https://www.youtube.com/watch?v=sjyLRS52zOg

▼ dmesg - Display kernel messages, such as boot messages, hardware-related messages, and error messages.

The `dmesg` command is a Linux/Unix utility that is used to display the kernel's ring buffer messages. The kernel ring buffer is a circular buffer that stores kernel messages, such as boot messages, hardware-related messages, and error messages.
The `dmesg` command displays the contents of the kernel ring buffer in a human-readable format.

When you execute the `dmesg` command, it displays the most recent kernel messages at the end of the ring buffer. You can use the `-c` option to clear the ring buffer before displaying the messages. This can be useful when you want to view only the new messages since the last time you ran `dmesg`.

Here is an example output of the `dmesg` command:

json

Copy

```
[    0.000000] Linux version 5.4.0-77-generic (buildd@lgw01-amd64-017) (gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #8
6-Ubuntu SMP Fri Apr 9 09:28:46 UTC 2021 (Ubuntu 5.4.0-77.86-generic 5.4.121)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.4.0-77-generic root=UUID=8abf7c5e-8d66-4d08-9691-2f6e36baf4f3 ro qui
et splash vt.handoff=7
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
[    0.000000]   AMD AuthenticAMD
[    0.000000]   Hygon HygonGenuine
[    0.000000]   Centaur CentaurHauls
[    0.000000]   zhaoxin   Shanghai
[    0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
```

▼ vmstat - display virtual memory statistics

The `vmstat` command is a Linux/Unix utility that is used to display virtual memory statistics. The output of the `vmstat` command provides information about system memory usage, including virtual memory, paging, swapping, and CPU usage.

When you execute the `vmstat` command, it displays a summary of system memory usage statistics at regular intervals. By default, the `vmstat` command displays the statistics every second. You can also specify a different interval by using the `-n` option followed by the number of seconds.

The `vmstat` command displays the following columns of statistics:

- `procs`: displays statistics about processes, such as the number of processes running, blocked, or waiting for I/O.
- `memory`: displays statistics about system memory usage, such as the amount of free, used, and buffered memory.
- `swap`: displays statistics about swap space usage, such as the amount of free, used, and cached swap space.
- `io`: displays statistics about input/output (I/O) operations, such as the number of blocks read and written to disk.
- `system`: displays statistics about system-level activity, such as interrupts per second and context switches per second.
- `cpu`: displays statistics about CPU usage, such as the percentage of time spent in user mode, system mode, and idle mode.

Here is an example output of the `vmstat` command:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0  53208  90556  10020 157364    0    0    39    89   43   94  9  2 89  0  0
 0  0  53208  90556  10020 157364    0    0     0    44  111  238  1  0 99  0  0
 0  0  53208  90556  10020 157364    0    0     0     0  103  220  1  0 99  0  0
 0  0  53208  90556  10020 157364    0    0     0     0  109  225  0  0 100  0  0
```

In this example, the `r` and `b` columns show the number of processes waiting to run and blocked on I/O operations, respectively. The `swpd`, `free`, `buff`, and `cache` columns show the amount of virtual memory used for swapped out processes, free memory, buffered memory, and cached memory, respectively. The `si` and `so` columns show the amount of data swapped in and out to disk, respectively. The `bi` and `bo` columns show the amount of data read from and written to disk, respectively. The `us`, `sy`, `id`, and `wa` columns show the percentage of CPU time spent in user mode, system mode, idle mode, and waiting for I/O operations, respectively. The `st` column shows the amount of time stolen from a virtual machine by the hypervisor for other tasks.

▼ System Information

▼ uname - [Unix Name] Display system information. (this is mycomputer properties option)

The uname command is a Linux/Unix command used to display system information about the **operating system and the hardware platform**. The name "uname" stands for "Unix Name", as it is a command used to display information about the underlying Unix/Linux operating system. It provides information such as the **kernel name, release version, and machine architecture.** The basic syntax of the uname command is as follows:

```
uname [option]
```

Here are some common options of the uname command:

1. a: Displays all system information.
2. s: Displays the **kernel** name.
3. r: Displays the **kernel release version**.
4. m: Displays the machine architecture.
5. o: Displays the operating system name.

Example Output:

1. **To display all system information:**

   ```
   uname -a
   ```

   This command displays all system information, including the kernel name, release version, machine architecture, and operating system name.

2. **To display the kernel name:**

   ```
   uname -s
   ```

   This command displays the kernel name, such as **"Linux" or "GNU".**

3. **To display the kernel release version:**

   ```
   uname -r
   ```

   This command displays the kernel release version, such as "**5.0.0-23-generic".**

4. **To display the machine architecture:**

   ```
   uname -m
   ```

   This command displays the machine architecture, such as **"x86_64" or "armv7l".**

5. To display the operating system name:

   ```
   uname -o
   ```

   This command displays the operating system name, such as "GNU/Linux" or "Android".

   In addition to these options, the uname command also supports other variations and options depending on the specific operating system and platform. For example, on some systems, the "`-i`" option can be used to display the hardware platform ID, while on others, the "`-p`" option can be used to display the processor type.

   In summary, the uname command is a useful tool for obtaining system information about the operating system and the hardware platform. Its various options allow for flexible and detailed analysis of system information, making it a valuable tool for system administrators and developers.

▼ uptime - Display system uptime.

The uptime command is a Linux/Unix command used to display the system's uptime, load average, and other system-related information. It provides information about **how long the system has been running**, the **number of users currently** logged in, and the average load on the system over different time intervals. The basic syntax of the uptime command is as follows:

```
uptime [option]
```

Here are some common options of the uptime command:

1. p: Displays the system's uptime in a more human-readable format.
2. s: Displays the time at which the system was last booted.
3. u: Displays the number of users currently logged in.

Example Output:

1. **To display the system's uptime:**

   ```
   uptime
   ```

   This command displays the system's uptime, load average, and the number of users currently logged in, in the following format:

   ```
    13:40:30 up 1 day,  2:14,  2 users,  load average: 0.00, 0.01, 0.05
   ```

   The output shows the current time, the system's uptime, the number of users currently logged in, and the average system load over the last 1, 5, and 15-minute intervals.

2. **To display the system's uptime in a more human-readable format:**

   ```
   uptime -p
   ```

   This command displays the system's uptime in a more human-readable format, showing the system's uptime in days, hours, and minutes. For example, the output may look like this:

   ```
   up 1 day,  2 hours,  14 minutes
   ```

3. **To display the time at which the system was last booted:**

   ```
   uptime -s
   ```

   This command displays the time at which the system was last booted, in the format of "YYYY-MM-DD HH:MM:SS". For example, the output may look like this:

   ```
   2022-03-27 13:40:30
   ```

4. **To display the number of users currently logged in:**

   ```
   uptime -u
   ```

   This command displays the number of users currently logged in to the system. For example, the output may look like this:

   ```
   2 users
   ```

   In addition to these options, the uptime command also supports other variations and options depending on the specific operating system and platform. For example, on some systems, the "-a" option can be used to display the system's boot time, and the "-i" option can be used to display the system's idle time.

   In summary, the uptime command is a simple yet useful tool for monitoring system uptime, load average, and other system-related information. Its various options allow for flexible and detailed analysis of system information, making it a valuable tool for system administrators and developers.

▼ free - Display System'memory usage. [like hard-disk usages]

The free command is a Linux/Unix command used to display information about the **system's memory usage and memory statistics.** It provides information about the total amount of memory available, the amount of memory used, and the amount of memory free. The basic syntax of the free command is as follows:

```
free [option]
```

Here are some common options of the free command:

1. b: Displays memory statistics in bytes.
2. k: Displays memory statistics in kilobytes.
3. m: Displays memory statistics in megabytes.
4. g: Displays memory statistics in gigabytes.
5. t: Displays a summary of total memory usage.

Example Output:

1. To display memory statistics in kilobytes:

```
free -k
```

This command displays the memory statistics in kilobytes, including the total amount of memory available, the amount of memory used, and the amount of memory free. The output may look like this:

```
              total        used        free      shared  buff/cache   available
Mem:         204800        4096      200704           0        4096      200704
Swap:             0           0           0
```

The output shows the total amount of memory, the amount of memory used, the amount of memory free, the amount of shared memory, the amount of memory used for buffering and caching, and the amount of memory available for new processes.

df - Report file system disk space usage [individual disk’s usage].

The df (disk free) command is a Linux/Unix command used to display information about the file system disk space usage. df provides information about the total disk space, used disk space, available disk space, and file system type. The basic syntax of the df command is as follows:

```
df [option]
```

Here are some common options of the df command:

1. h: Displays disk usage in a more human-readable format.
2. T: Displays the file system type.
3. i: Displays the number of inodes used and available.
4. t: Displays information for a specific file system type.
5. x: Excludes specific file system types.

Example Output:

1. To display disk usage in a more human-readable format:

```
df -h
```

This command displays the disk usage in a more human-readable format, showing the total disk space, used disk space, available disk space, and file system type. The output may look like this:

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        20G   13G  5.7G  71% /
/dev/sda2        30G  7.8G   20G  29% /home
```

The output shows the total size of the file system, the amount of space used, the amount of space available, the percentage of space used, and the mount point of the file system.

df vs free

`df` and `free` are two commands that are used to look at information about the space and memory on your computer. `df` tells you how much space is used and available on your **computer's different file systems**. `free` tells you how much memory is used and available in your computer's system.

```
girish@girish-Inspiron-5570:~/Downloads/umask$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           785M  2.5M  783M   1% /run
/dev/sda6        97G   19G   73G  21% /
tmpfs           3.9G   59M  3.8G   2% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
/dev/sdb1       646M  107M  540M  17% /boot/efi
tmpfs           785M  7.6M  777M   1% /run/user/1000
girish@girish-Inspiron-5570:~/Downloads/umask$ free -k
               total        used        free      shared  buff/cache   available
Mem:         8033392     4380348      295628     1809700     3357416     1529684
```

```
Swap:        9764860     1395252     8369608
girish@girish-Inspiron-5570:~/Downloads/umask$
```

▼ what are the different file systems

There are several different types of file systems. Common types include NTFS, FAT32, exFAT, ext2/ext3/ext4, HFS+, and APFS. Each type of file system has its own advantages and disadvantages.

▼ du - Estimate file space usage. [like right-click and properties]

The `du` (disk usage) command is a Linux/Unix command used to estimate **disk space usage for files and directories.** It provides information about the amount of disk space used by each file and directory in a given directory tree. The basic syntax of the `du` command is as follows:

```
du [options] [directories/files]
```

Here are some common options of the `du` command:

1. `h`: Displays disk usage in a more human-readable format.
2. `s`: Displays only a summary of the disk usage for each directory.
3. `c`: Displays a grand total of the disk usage for all directories.
4. `a`: Displays disk usage for all files, including hidden files.
5. `x`: Excludes files on different file systems from being included in the output.

Example Output:

1. To display disk usage in a more human-readable format:

```
du -h
```

This command displays the disk usage in a more human-readable format, showing the disk space used by each file and directory in the current directory. The output may look like this:

```
4.0K  ./dir1
8.0K  ./dir2
12K ./file1.txt
20K .
```

The output shows the amount of disk space used by each file and directory in the current directory, in a human-readable format.

▼ tty - Print the file name of the terminal connected to standard input.

The `tty` command is a Linux/Unix command used to display the name of the terminal device connected to the standard input. It provides information about the terminal device being used by the current shell or process. The basic syntax of the `tty` command is as follows:

```
tty [options]
```

Here are some common options of the `tty` command:

1. `s`: Suppresses the output of the filename, so only the device name is displayed.

Example Output:

1. To display the name of the terminal device connected to the standard input:

```
tty
```

This command displays the name of the terminal device connected to the standard input. The output may look like this:

```
/dev/pts/1
```

The output shows the name of the terminal device being used by the current shell or process. In this case, the terminal device is `/dev/pts/1`, which is a **pseudo-terminal.**

▼ ulimit - Set or display resource limits.

The `ulimit` command is a Linux/Unix command used to set or display resource limits for the current shell or process. It provides a way to control and manage system resources such as memory, CPU time, file size, and more. The basic syntax of the `ulimit` command is as follows:

```
ulimit [-H|-S] [resource] [value]
```

Here are some common options of the `ulimit` command:

1. `H`: Sets or displays the hard limit for the specified resource.
2. `S`: Sets or displays the soft limit for the specified resource.

Here are some common resources that can be controlled using the `ulimit` command:

1. `core`: Maximum size of a core file (a file containing a program's memory dump in case of a crash).
2. `cpu`: Maximum CPU time in seconds.
3. `data`: Maximum data segment size (heap and stack combined) in bytes.
4. `fsize`: Maximum size of a file created by the current process in bytes.
5. `nofile`: Maximum number of open file descriptors (files, sockets, devices, etc.).
6. `nproc`: Maximum number of processes per user.

Example Output:

1. To display the current resource limits for the current shell or process:

   ```
   ulimit -a
   ```

   This command displays the current resource limits for the current shell or process. The output may look like this:

   ```
   core file size          (blocks, -c) 0
   data seg size           (kbytes, -d) unlimited
   file size               (blocks, -f) unlimited
   max locked memory       (kbytes, -l) unlimited
   max memory size         (kbytes, -m) unlimited
   open files                      (-n) 1024
   pipe size            (512 bytes, -p) 8
   stack size              (kbytes, -s) 8192
   cpu time               (seconds, -t) unlimited
   max user processes              (-u) 1024
   virtual memory          (kbytes, -v) unlimited
   ```

   The output shows the current resource limits for various resources, such as the maximum size of core files, data segment size, file size, maximum memory size, maximum number of open files, and more.

2. To set the maximum number of open files to 2048:

   ```
   ulimit -n 2048
   ```

   This command sets the maximum number of open files to 2048. The output may look like this:

   ```
   (none)
   ```

   The output does not provide any confirmation message, but the limit is set and can be verified by running the `ulimit -a` command.

3. To set the soft limit for the maximum CPU time to 2 minutes:

```
ulimit -t 120
```

This command sets the soft limit for the maximum CPU time to 2 minutes (120 seconds). The output may look like this:

```
(none)
```

The output does not provide any confirmation message, but the limit is set and can be verified by running the `ulimit -a` command.

In summary, the `ulimit` command is a powerful tool for controlling and managing system resources for the current shell or process. Its various options and resources allow for flexible and detailed management of system resources, making it a valuable tool for system administrators and developers. However, care should be taken when setting resource limits, as improper usage can cause system instability or performance issues.

User Management

who - Display currently logged in users.

The `who` command is a Linux/Unix command used to display information about currently logged-in users. It provides information about the username, terminal or device used to log in, the time of login, and the IP address or hostname of the remote host. The basic syntax of the `who` command is as follows:

```
who [options] [filename]
```

Here are some common options of the `who` command:

1. `a`: Displays all information, including the hostname of the remote host.
2. `b`: Displays the time when the system was last rebooted.
3. `r`: Displays the current runlevel of the system.
4. `u`: Displays the username and idle time of each logged-in user.
5. `H`: Displays the column headings for the output.

Example Output:

1. To display information about currently logged-in users:

```
who
```

This command displays information about currently logged-in users. The output may look like this:

```
username  tty7         2022-03-25 15:30 (:0)
username  pts/0        2022-03-26 10:15 (192.168.1.2)
```

The output shows the username, terminal or device used to log in, the time of login, and the IP address or hostname of the remote host.

last - Display last logged in users.

The `last` command is a Linux/Unix command used to display information about the last logged-in users on the system. It provides information about the username, terminal or device used to log in, the time of login, and the duration of the login session. The basic syntax of the `last` command is as follows:

```
last [options] [username]
```

Here are some common options of the `last` command:

1. `n`: Specifies the number of login records to display.

2. `a`: Displays the hostname in the last column of the output.
3. `f`: Displays the full date and time of the login instead of the abbreviated format.
4. `i`: Displays IP addresses instead of hostnames.
5. `t`: Specifies the start time for the search.

Example Output:

1. To display information about the last logged-in users:

```
last
```

This command displays information about the last logged-in users on the system. The output may look like this:

```
username  pts/0        hostname   Fri Jul 23 12:30   still logged in
username  pts/0        hostname   Fri Jul 23 10:15 - 11:00  (00:45)
username  pts/0        hostname   Fri Jul 23 09:30 - 09:45  (00:15)
```

The output shows the username, terminal or device used to log in, the time of login, and the duration of the login session.

▼ id - Display user and group information.

The `id` command is a Linux/Unix command used to display user and group identity information for the current user or a specified user. It provides information about the numeric user ID (UID), numeric group ID (GID), and supplementary group IDs of the current user or a specified user. The basic syntax of the `id` command is as follows:

```
id [options] [username]
```

Here are some common options of the `id` command:

1. `u`: Displays only the numeric user ID (UID) of the current user or a specified user.
2. `g`: Displays only the numeric group ID (GID) of the current user or a specified user.
3. `n`: Displays the username associated with the UID or GID.
4. `G`: Displays the supplementary group IDs of the current user or a specified user, separated by commas.

Example Output:

1. To display the UID and GID of the current user:

```
id
```

This command displays the UID and GID of the current user. The output may look like this:

```
uid=1000(username) gid=1000(groupname) groups=1000(groupname),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126
(sambashare)
```

The output shows the UID, GID, and supplementary group IDs of the current user.

▼ w - Display logged in users and their processes.**

The `w` command is a Linux/Unix command used to display information about currently logged-in users and their processes. It provides information about the username, terminal or device used to log in, the time of login, the IP address or hostname of the remote host, and the list of currently running processes for each logged-in user. The basic syntax of the `w` command is as follows:

```
w [options] [username]
```

Here are some common options of the `w` command:

1. `a`: Displays information about all logged-in users and their processes.
2. `h`: Suppresses the display of the column headings.

3. `s`: Displays only the current system load average.
4. `u`: Displays detailed information about each logged-in user and their processes.

Example Output:

1. To display information about currently logged-in users and their processes:

```
w
```

This command displays information about currently logged-in users and their processes. The output may look like this:

```
 09:30:00 up 10 days,  1:23,  3 users,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
username pts/0    192.168.1.2      09:30    0.00s  0.01s  0.00s w
username pts/1    192.168.1.3      08:15    1:23m  0.05s  0.05s -bash
username pts/2    192.168.1.4      07:45    2.00s  0.03s  0.02s top
```

The output shows the username, terminal or device used to log in, the time of login, the IP address or hostname of the remote host, and the list of currently running processes for each logged-in user.

▼ users - Display logged in users.

The `users` command is a Linux/Unix command used to display a list of currently logged-in users. It provides a simple output that lists the usernames of all currently logged-in users in a space-separated format. The basic syntax of the `users` command is as follows:

```
users [options]
```

Here are some common options of the `users` command:

1. `q`: Displays only the number of currently logged-in users.
2. `s`: Displays the time when the system was last booted.

Example Output:

1. To display a list of currently logged-in users:

```
users
```

This command displays a list of currently logged-in users. The output may look like this:

```
username1 username2 username3
```

The output shows the usernames of all currently logged-in users, separated by spaces.

Who vs users

```
girish@girish-Inspiron-5570:~/Downloads/umask$ who
girish   tty2         2023-04-29 11:04 (tty2)
girish@girish-Inspiron-5570:~/Downloads/umask$ users
girish
```

▼ su - Switch user or become superuser.

The su (short for "substitute user") command is used to switch to another user account in Linux. It is commonly used to run commands with elevated privileges, such as when performing system administration tasks.

The basic syntax for the su command is as follows:

```
su [options] [username]
```

where `username` is the name of the user account to switch to. If no username is specified, the command will switch to the root user account by default.

Here is an example of how to use the su command:

```
su -
```

This command will switch to the **root user account** and start a new login shell. The `-` option specifies that a login shell should be started, which initializes the environment variables and settings for the new user account.

The possible output for this example is as follows:

```
Password:
```

This output prompts the user to enter the password for the root user account. Once the password is entered and authenticated, the user will be switched to the root user account and a new login shell will be started.

Here is another example, where the su command is used to switch to a specific user account:

```
su - john
```

This command will switch to the user account **named `john` and start a new login shell.**

The possible output for this example is the same as the previous example, which prompts the user to enter the password for the specified user account. Once the password is entered and authenticated, the user will be switched to the `john` user account and a new login shell will be started.

The su command is a powerful tool that should be used with caution, as it grants access to sensitive system resources and can potentially cause damage if used improperly. It is important to only switch to user accounts that are necessary for the task at hand and to log out of the account when the task is complete.

useradd - Used to create a new user account.

The useradd command is used to create a new user account in Linux. It is typically used by system administrators to create user accounts for new users or to **create service accounts for specific applications.**

The basic syntax for the useradd command is as follows:

```
useradd [options] username
```

where `username` is the name of the user account to create.

Here are some examples of how to use the useradd command:

1. Create a new user account with default settings:

   ```
   useradd john
   ```

   This command will create a new user account named `john` with default settings. The user's home directory will be created in the /home directory, and the user's login shell will be set to /bin/bash. The password for this method should be setup using the `passwd` command.

   > 💡 **By default, when you create a user using the "useradd" command in Linux, a new user account is created with a unique user ID (UID) and a new group is also created with the same name as the user. The new user is then added as a member of this group.**

2. Create a new user account with a **custom home directory and login shell:**

   ```
   useradd -d /data/john -s /bin/zsh john
   ```

This command will create a new user account named `john` with a custom home directory (/data/john) and a custom login shell (/bin/zsh).

The possible output for this example is the same as the previous example, which indicates that the user account was created successfully.

3. Create a new user account with a custom **UID and GID:**

```
useradd -u 1001 -g 1001 john
```

This command will create a new user account named `john` with a custom UID (1001) and GID (1001).

The possible output for this example is the same as the previous examples, which indicates that the user account was created successfully.

4. Create a new user account with a custom password:

```
useradd -p 'password' john
```

This command will create a new user account named `john` with a custom password ('password').

The possible output for this example is the same as the previous examples, which indicates that the user account was created successfully.

Note: It is not recommended to set the password using the -p option, as it stores the password in plaintext in the /etc/shadow file. It is recommended to use the passwd command to set the password after creating the user account.

There are many other options and variations of the useradd command, including the ability to specify additional user information, set user account expiration dates, and create system accounts. By mastering the useradd command, you can create and manage user accounts efficiently and effectively on your Linux system.

command to check the users present in linux

In Linux, you can use the "cat" command to view the "/etc/passwd" file, which contains a list of all the user accounts on the system. Each line in the file represents one user account and includes information such as the username, user ID, home directory, and default shell.

To check the users present in Linux, you can run the following command:

```
cat /etc/passwd
```

This command will display the contents of the "/etc/passwd" file in the terminal, showing a list of all user accounts on the system.

```
[ec2-user@ip-172-31-35-68 ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
.
.
.
.

tcpdump:x:72:72::/:/sbin/nologin
ec2-user:x:1000:1000:EC2 Default User:/home/ec2-user:/bin/bash
john:x:1001:1001::/home/john:/bin/bash
ram:x:1002:1002::/home/ram:/bin/bash
seeta:x:1003:1003::/home/seeta:/bin/bash
```

You can also use the "cut" command to display only the usernames of the users present in Linux. To do this, you can run the following command:

```
cut -d: -f1 /etc/passwd
```

This command will display a list of only the usernames, separated by line breaks.

▼ userdel - Used to delete a user account and its files

The "userdel" command in Linux is used to delete a user account and its associated files from the system. Here are some variations of the "userdel" command with examples and possible output for each example:

1. Deleting a user account:

   ```
   sudo userdel username
   ```

   This command deletes the "username" user account and its home directory and mail spool. The possible output could look like this:

   ```
   userdel: user 'username' removed
   ```

   In this example, the "username" user account is deleted, and the output confirms that the account has been removed.

2. Deleting a user account and **its files:**

   ```
   sudo userdel -r username
   ```

   This command deletes the "username" user account and its home directory and mail spool. The "-r" option removes the user's home directory and mail spool along with the account. The possible output could look like this:

   ```
   userdel: user 'username' removed
   ```

   In this example, the "username" user account and its files are deleted, and the output confirms that the account has been removed.

3. Deleting a user account and **its group:**

   ```
   sudo userdel -r -f username
   ```

   This command deletes the "username" user account and its home directory and mail spool, as well as the user's primary group. The "-f" option forces the removal of the user's files, even if the files are not owned by the user. The possible output could look like this:

   ```
   userdel: user 'username' removed
   ```

   In this example, the "username" user account, its files, and its primary group are deleted, and the output confirms that the account has been removed.

4. Removing a user account and specifying **a backup directory:**

   ```
   sudo userdel -r -b /backup username
   ```

   This command deletes the "username" user account and its home directory and mail spool, and backs up the user's files to the "/backup" directory. The "-b" option specifies the backup directory. The possible output could look like this:

   ```
   userdel: user 'username' removed
   ```

   In this example, the "username" user account and its files are deleted, and the output confirms that the account has been removed.

   Overall, the "userdel" command is a useful tool for deleting user accounts and their associated files from a Linux system. The options and variations of the command provide flexibility for customizing the deletion process. It's important to be careful when using the "userdel" command, as it permanently deletes files and cannot be undone.

▼ passwd - Used to change the password for a user account.

The "passwd" command in Linux is used to change the password of a user account. Here are some variations of the "passwd" command with examples and possible output for each example:

1. Changing the password of the **current user:**

   ```
   passwd
   ```

   This command prompts the current user to enter their current password, then enter a new password twice for verification. The possible output could look like this:

   ```
   Changing password for user username.
   (current) UNIX password:
   New UNIX password:
   Retype new UNIX password:
   passwd: password updated successfully
   ```

   In this example, the password for the current user is changed, and the output confirms that the password was updated successfully.

2. Changing the password of another user:

   ```
   sudo passwd username
   ```

   This command prompts the user with sudo privileges to enter a new password for the "username" user account. The possible output could look like this:

   ```
   Enter new UNIX password:
   Retype new UNIX password:
   passwd: password updated successfully
   ```

   In this example, the password for the "username" user account is changed, and the output confirms that the password was updated successfully.

3. Specifying a new password without prompting:

   ```
   echo "newpassword" | passwd --stdin username
   ```

   This command changes the password for the "username" user account to "newpassword" without prompting the user to enter the password. The "--stdin" option tells the "passwd" command to read the new password from standard input. The possible output could look like this:

   ```
   passwd: password updated successfully
   ```

   In this example, the password for the "username" user account is changed to "newpassword", and the output confirms that the password was updated successfully.

4. Forcing the user to change their password on next login:

   ```
   sudo passwd -e username
   ```

   This command sets the "username" user account to expire, forcing the user to change their password on their next login. The "-e" option sets the account expiration date to January 1, 1970, which marks the account as expired. The possible output could look like this:

   ```
   passwd: password expiry information changed
   ```

   In this example, the "username" user account is set to expire, and the output confirms that the password expiry information was changed.

Overall, the "passwd" command is a useful tool for changing the password of a user account in Linux. The options and variations of the command provide flexibility for customizing the password change process. It's important to follow best practices for password management and security, such as using strong passwords and changing them regularly.

▼ chpasswd - Used to change the passwords for multiple user accounts at once.

The chpasswd command is used to change the password for **multiple user accounts at once in Linux.** It takes its input from a text file or from standard input, and it can be used to change the passwords for a large number of users in a single command.

The basic syntax for the chpasswd command is as follows:

```
chpasswd [options]
```

The passwords for the user accounts are provided in the following format:

```
username:password
```

where `username` is the name of the user account and `password` is the new password for the user account.

Here are some examples of how to use the chpasswd command:

1. Change the password for a single user account:

   ```
   echo 'john:newpassword' | chpasswd
   ```

   This command will change the password for the user account `john` to `newpassword`. The `echo` command is used to provide the input to the chpasswd command in the required format.

   The possible output for this example is as follows:

   ```
   
   ```

   This output indicates that the password was changed successfully, with no error messages or warnings.

2. Change the passwords for multiple user accounts using a file:

   ```
   cat passwords.txt | chpasswd
   ```

   This command will change the passwords for the user accounts listed in the `passwords.txt` file. The file should contain one line for each user account, in the format `username:password`.

   The possible output for this example is the same as the previous example, which indicates that the passwords were changed successfully.

3. Change the passwords for multiple user accounts interactively:

   ```
   chpasswd -i
   ```

   This command will prompt the user to enter the username and new password for each user account, one at a time. This can be useful for changing passwords for a small number of user accounts.

   The possible output for this example is as follows:

   ```
   Changing password for user john.
   New password:
   Retype new password:
   passwd: all authentication tokens updated successfully.
   ```

   This output shows the prompts for entering the new password and the confirmation of the password change.

   There are many other options and variations of the chpasswd command, including the ability to specify a different password file, encrypt passwords using different encryption algorithms, and change the password for a user account without requiring the old password. By mastering the chpasswd command, you can change passwords for multiple user accounts quickly and efficiently on your Linux system.

▼ usermod - Used to modify an existing user account.

The usermod command is used to modify an existing user account in Linux. It is typically used by system administrators to change user account settings, such as the user's home directory, login shell, or group membership.

The basic syntax for the usermod command is as follows:

```
usermod [options] username
```

where `username` is the name of the user account to modify.

Here are some examples of how to use the usermod command:

1. **Change the user's home directory:**

   ```
   usermod -d /data/john john
   ```

   This command will change the home directory for the user account `john` to `/data/john`.

   The possible output for this example is as follows:

   ```
   ```

   This output indicates that the user account was modified successfully, with no error messages or warnings.

2. **Change the user's login shell:**

   ```
   usermod -s /bin/zsh john
   ```

   This command will change the login shell for the user account `john` to `/bin/zsh`.

   The possible output for this example is the same as the previous example, which indicates that the user account was modified successfully.

3. **Add the user to a new group:**

   ```
   usermod -aG developers john
   ```

   This command will add the user account `john` to the `developers` group. ***The group must present already***

   The possible output for this example is the same as the previous examples, which indicates that the user account was modified successfully.

   Note: The `-a` option is used to append the **new group to the user's existing groups, rather than replacing them**.

4. **Remove the user from a group:**

   ```
   usermod -G developers john
   ```

   This command will remove the user account `john` from the `developers` group.

   The possible output for this example is the same as the previous examples, which indicates that the user account was modified successfully.

   There are many other options and variations of the usermod command, including the ability to set user account expiration dates, change the user's primary group, and lock or unlock a user account. By mastering the usermod command, you can modify user accounts efficiently and effectively on your Linux system.

▼ groupadd -

In Linux, you can use the following commands to create, assign a user to a group, and delete groups:

1. **Creating a group:**

```
sudo groupadd groupname
```

This command creates a new group with the name "groupname". The possible output could look like this:

```
groupadd: group 'groupname' added
```

In this example, the "groupname" group is created, and the output confirms that the group was added.

2. **Assigning a user to a group:**

```
sudo usermod -aG groupname username
```

This command adds the "username" user to the "groupname" group. The "-aG" option appends the user to the group without removing them from other groups, and the "-G" option specifies the group to add the user to. The possible output could look like this:

```
usermod: user 'username' appended to group 'groupname'
```

In this example, the "username" user is added to the "groupname" group, and the output confirms that the user was appended to the group.

3. **Deleting a group:**

```
sudo groupdel groupname
```

This command deletes the "groupname" group. The possible output could look like this:

```
groupdel: group 'groupname' removed
```

In this example, the "groupname" group is deleted, and the output confirms that the group was removed.

Note that you need superuser privileges (sudo) to create and delete groups, and to add or remove users from groups. It's important to follow best practices for group management and security, such as limiting group membership to only what is necessary for each user's role, and avoiding giving unnecessary privileges to users.

▼ how to create a user and assign it to a new group together

To create a user and assign it to a new group at the same time in Linux, you can use the following command:

```
sudo useradd -G groupname username
```

In this command, the "-G" option specifies the name of the new group that the user should be added to, and the "username" parameter specifies the username of the new user.

You can verify that the user and group were created by viewing the "/etc/passwd" and "**/etc/group**" files using the "cat" command or a text editor, as described in the previous answer.

▼ groups - Used to display the groups that a user account belongs to.

The "groups" command in Linux is used to display the group memberships of a user. Here are some variations of the "groups" command with examples and possible output for each example:

1. **Displaying the groups of the current user:**

```
groups
```

This command displays a list of the groups that the current user is a member of. The possible output could look like this:

```
username adm cdrom sudo dip plugdev lpadmin sambashare
```

In this example, the current user is a member of several groups, as listed in the output.

2. **Displaying the groups of another user:**

```
groups username
```

This command displays a list of the groups that the "username" user is a member of. The possible output could look like this:

```
username : username adm cdrom sudo dip plugdev lpadmin sambashare
```

In this example, the "username" user is a member of several groups, as listed in the output.

Overall, the "groups" command is a useful tool for displaying the group memberships of a user in Linux. The options and variations of the command provide flexibility for customizing the output and displaying group information for processes. It's important to understand group membership and management in Linux, as it plays a critical role in access control and system security.

▼ whoami - Used to display the username of the current user.

The "whoami" command in Linux is used to display the username of the current user. Here is an example of how to use the "whoami" command:

```
whoami
```

When you run this command, it will display the username of the current user in the terminal. For example, if the current user is "john", the output of the command will be:

```
john
```

This command can be **useful in scripts or commands where you need to reference** the current user's username, such as to check permissions or perform actions based on the current user's identity.

▼ logname - Used to display the login name of the current user.**

The "logname" command in Linux is used to display the login name of the current user. Here is an example of how to use the "logname" command:

```
logname
```

When you run this command, it will display the login name of the current user in the terminal. For example, if the current user is "john", the output of the command will be:

```
john
```

This command can be useful in scripts or commands where you need to reference the login name of the current user, such as to check permissions or perform actions based on the current user's identity.

Note that the "logname" command only displays the login name of the current user and does not provide any additional information about the user or their account. To view more detailed information about a user account, you can use the "whoami" command to display the username or the "id" command to display the UID, GID, and group memberships of a user.

▼ Network Management

▼ ifconfig - Configure network interfaces.

The `ifconfig` command is a Linux/Unix command used to display and configure network interfaces. It provides a way to view and change network interface configurations such as IP address, netmask, broadcast address, and more. The basic syntax of the `ifconfig` command is as follows:

```
ifconfig [interface] [options]
```

Here are some common options of the `ifconfig` command:

1. `up`: Brings the specified interface up.
2. `down`: Brings the specified interface down.
3. `inet`: Specifies the IP address and netmask for the interface.
4. `broadcast`: Specifies the broadcast address for the interface.

Example Output:

1. To display information for all network interfaces:

```
ifconfig
```

This command displays information for all network interfaces. The output may look like this:

```
eth0      Link encap:Ethernet  HWaddr 00:11:22:33:44:55
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::211:22ff:fe33:4455/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:263550 errors:0 dropped:0 overruns:0 frame:0
          TX packets:138111 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:36083966 (36.0 MB)  TX bytes:14154122 (14.1 MB)
```

The output shows information for the `eth0` network interface, including its hardware address, IP address, netmask, broadcast address, and more.

▼ netstat - Display network connections.

The `netstat` command is a Linux/Unix command used to display network connections and statistics. It provides a way to view various network-related information such as active connections, listening sockets, routing tables, and more. The basic syntax of the `netstat` command is as follows:

```
netstat [options]
```

Here are some common options of the `netstat` command:

1. `a`: Displays all sockets (both listening and non-listening).
2. `n`: Displays numerical addresses instead of resolving hostnames.
3. `t`: Displays TCP connections.
4. `u`: Displays UDP connections.
5. `r`: Displays routing tables.

Example Output:

1. To display all active network connections:

```
netstat -a
```

This command displays all active network connections, both listening and non-listening. The output may look like this:

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 192.168.1.100:ssh       192.168.1.200:12345     ESTABLISHED
tcp        0      0 192.168.1.100:http      192.168.1.200:45678     TIME_WAIT
udp        0      0 192.168.1.100:syslog    0.0.0.0:*
```

The output shows active network connections and their states, including the protocol, local and foreign addresses, and more.

1. To display numerical addresses for active network connections:

   ```
   netstat -n
   ```

   This command displays numerical addresses instead of resolving hostnames for active network connections. The output may look like this:

   ```
   Active Internet connections (servers and established)
   Proto Recv-Q Send-Q Local Address           Foreign Address         State
   tcp        0      0 192.168.1.100:22        192.168.1.200:12345     ESTABLISHED
   tcp        0      0 192.168.1.100:80        192.168.1.200:45678     TIME_WAIT
   udp        0      0 192.168.1.100:514       0.0.0.0:*
   ```

   The output shows numerical addresses for active network connections, making it easier to read and analyze complex network configurations.

2. To display TCP connections only:

   ```
   netstat -t
   ```

   This command displays TCP connections only. The output may look like this:

   ```
   Active Internet connections (w/o servers)
   Proto Recv-Q Send-Q Local Address           Foreign Address         State
   tcp        0      0 192.168.1.100:ssh       192.168.1.200:12345     ESTABLISHED
   tcp        0      0 192.168.1.100:http      192.168.1.200:45678     TIME_WAIT
   ```

   The output shows TCP connections only, making it easier to focus on a specific type of network traffic.

3. To display routing tables:

   ```
   netstat -r
   ```

   This command displays routing tables for the system. The output may look like this:

   ```
   Kernel IP routing table
   Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
   default         192.168.1.1     0.0.0.0         UG        0 0          0 eth0
   10.0.0.0        0.0.0.0         255.255.255.0   U         0 0          0 eth1
   192.168.1.0     0.0.0.0         255.255.255.0   U         0 0          0 eth0
   ```

   The output shows the routing tables for the system, including destination addresses, gateways, netmasks, and more.

   In summary, the `netstat` command is a powerful tool for displaying and analyzing network connections and statistics on a Linux/Unix system. Its various options and configurations allow for flexible and detailed network analysis, making it a valuable tool for system administrators and network engineers.

▼ traceroute - Trace the path to a host.

The "traceroute" command is a network diagnostic tool used to track the path of packets as they travel from one network device to another. It works by sending packets with incrementing TTL (Time-to-Live) values and recording the IP addresses of the routers that the packets pass through. Here are some variations of the "traceroute" command with examples:

1. "traceroute [destination]": This command will display the route that packets take to reach the specified destination.

Example command: `traceroute google.com`

Example output:

```
traceroute to google.com (172.217.7.78), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.634 ms  1.299 ms  1.361 ms
 2  10.0.0.1 (10.0.0.1)  4.051 ms  4.491 ms  4.213 ms
 3  172.16.0.1 (172.16.0.1)  5.433 ms  5.141 ms  5.416 ms
 4  203.131.240.1 (203.131.240.1)  12.926 ms  12.602 ms  12.878 ms
 5  203.131.246.30 (203.131.246.30)  13.178 ms  13.470 ms  13.155 ms
```

```
 6  203.131.246.81 (203.131.246.81)  12.968 ms  13.233 ms  13.067 ms
 7  203.131.246.124 (203.131.246.124)  13.332 ms  12.862 ms  13.625 ms
 8  72.14.212.215 (72.14.212.215)  16.546 ms  17.019 ms  16.702 ms
 9  108.170.251.209 (108.170.251.209)  17.300 ms  17.269 ms  17.310 ms
10  172.217.7.78 (172.217.7.78)  16.960 ms  16.965 ms  16.659 ms
```

1. "traceroute -I [destination]": This command will use ICMP echo requests instead of UDP packets to trace the route.

Example command: `traceroute -I google.com`

Example output:

```
traceroute to google.com (172.217.7.78), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.634 ms  1.299 ms  1.361 ms
 2  10.0.0.1 (10.0.0.1)  4.051 ms  4.491 ms  4.213 ms
 3  172.16.0.1 (172.16.0.1)  5.433 ms  5.141 ms  5.416 ms
 4  203.131.240.1 (203.131.240.1)  12.926 ms  12.602 ms  12.878 ms
 5  203.131.246.30 (203.131.246.30)  13.178 ms  13.470 ms  13.155 ms
 6  203.131.246.81 (203.131.246.81)  12.968 ms  13.233 ms  13.067 ms
 7  203.131.246.124 (203.131.246.124)  13.332 ms  12.862 ms  13.625 ms
 8  72.14.212.215 (72.14.212.215)  16.546 ms  17.019 ms  16.702 ms
 9  108.170.251.209 (108.170.251.209)  17.300 ms  17.269 ms  17.310 ms
10  172.217.7.78 (172.217.7.78)  16.960 ms  16.965 ms  16.659 ms
```

▼ ssh - Secure remote login.

SSH (Secure Shell) is a cryptographic network protocol used to securely access and manage network devices and servers over an unsecured network. As a DevOps engineer, understanding SSH is crucial for securely managing and automating tasks in Linux environments. Here's a comprehensive overview of SSH for DevOps engineers:

Overview

- **Protocol**: SSH is a protocol that runs on top of TCP/IP, using port 22 by default.
- **Encryption**: SSH uses public key cryptography to ensure secure communication between client and server.
- **Authentication**: SSH supports various types of authentication, including **password-based, public key-based, and multi-factor authentication.**
- **Applications**: SSH provides secure channel for remote login, command execution, file transfer, and port forwarding.

Components

1. **SSH Client**: The software used to initiate and manage SSH connections from the **user's machine.**
2. **SSH Server**: The software running on the remote machine that accepts and manages **incoming SSH connections**.
3. **SSH Key Pair**: A pair of cryptographic keys (public and private) used for authentication and encryption.

   ▼ what is the difference between ssh public and private key

   SSH (Secure Shell) is a protocol used for secure remote access to a server or computer. It uses a pair of keys, a public key, and a private key, to authenticate and encrypt communication between two devices. Here's the difference between SSH public and private keys:

   1. Public Key: A public key is a key that is shared with other devices or servers to allow them to communicate with each other securely. It is used to encrypt data that is sent to the device or server. The public key is usually stored on the server, and anyone who has access to it can send encrypted data that only the server can decrypt with its corresponding private key. The public key can be shared freely, as it cannot be used to decrypt data.
   2. Private Key: A private key is a key that is kept secret and is known only to the owner of the key. It is used to decrypt data that is encrypted with its corresponding public key. The private key is usually stored on the client device, and it should never be shared with anyone else. If someone gains access to the private key, they can read all the encrypted data sent to the client.

   In summary, the public key is used to encrypt data, and the private key is used to decrypt it. The public key can be shared freely, while the private key should be kept secret. Both keys are necessary for secure communication between two devices using SSH.

   explain like i am ten

When you want to talk to a friend on the phone, you need to make sure that only your friend can hear you and nobody else can listen in. You can do this by using a secret code that only you and your friend know. In the same way, when you want to talk to a computer or server through SSH, you need to make sure that only the computer can hear you and nobody else can listen in.

To do this, you use two keys called the public key and the private key. Think of the public key as a lock and the private key as a key that fits into that lock. You give the lock (public key) to the computer or server, and you keep the key (private key) with you. When you want to talk to the computer or server, you use the key (private key) to unlock the lock (public key) and start talking.

The lock (public key) can't be opened with the key (private key), so it's safe to share it with anyone. But the key (private key) is special and should always be kept secret because anyone who gets it can use it to talk to the computer or server and read all the messages you send.

The public key is used to **encrypt the data** that is sent to the server, while the private key **is used to decrypt it.** The public key can be shared freely with anyone who wants to communicate with the server. The private key, on the other hand, is kept secret and should never be shared with anyone.

Key Concepts

Key Generation

To generate an SSH key pair, use the `ssh-keygen` command:

```
ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
```

Here's a breakdown of the command and its options:

- "ssh-keygen": This is the command itself that will generate the SSH keys.
- "-t rsa": This option specifies the type of key to be generated. In this case, it's an RSA key, which is a popular type of public-key cryptography.
- "-b 4096": This option specifies the length of the key in bits. In this case, it's 4096 bits, which is considered to be a very secure key length.
- "-C "your.email@example.com"": This option is used to add a comment to the key. The comment is used to identify the key owner or provide any additional information about the key. In this case, it's your email address.

Public Key Authentication

To use public key authentication, copy the public key to the remote server's `~/.ssh/authorized_keys` file:

```
ssh-copy-id user@remote-host
```

▼ SSH Config File

The SSH config file is a powerful tool that allows you to customize your SSH connections and automate the process of connecting to remote servers. Here are some use cases of the SSH config file:

1. Customizing connection parameters: You can use the SSH config file to customize the connection parameters for individual servers. For example, you can set the username, hostname, and port number for each server in the config file, so you don't have to provide them every time you connect to the server.
2. Specifying the private key location: You can use the SSH config file to specify the location of the private key used for authentication. This can be helpful if you have multiple private keys and want to use a specific one for a particular server.
3. Setting up port forwarding: You can use the SSH config file to set up port forwarding, which allows you to forward traffic from one port on your local machine to another port on a remote server.

4. Automating logins: You can use the SSH config file to automate the login process. This can be helpful if you have to connect to a server frequently and don't want to enter your password every time.
5. Setting up ProxyJump: You can use the SSH config file to set up ProxyJump, which allows you to connect to a remote server through an intermediate server. This can be helpful if you need to connect to a server that is not directly accessible from your local machine.
6. Defining aliases: You can use the SSH config file to define aliases for commonly used commands. For example, you can define an alias for "ssh example.com" as "ssh myserver" in the config file, so you can simply type "ssh myserver" to connect to "example.com".

Overall, the SSH config file is a versatile tool that allows you to customize your SSH connections and automate the process of connecting to remote servers, making your life much easier.

The SSH config file is a configuration file used to set up SSH connections and customize their behavior. The file is usually located in the ~/.ssh directory and is named "config". Here are some variations of the SSH config file with examples:

1. Basic configuration: Here's an example of a basic SSH config file:

   ```
   Host example.com
     User username
     Hostname example.com
     IdentityFile ~/.ssh/id_rsa
   ```

   This configuration sets up a connection to a server named "example.com" and specifies the username to use when connecting to the server. It also specifies the location of the private key that should be used to authenticate the connection.

2. Port forwarding: Port forwarding is a technique used to forward traffic from one port on a local machine to another port on a remote machine. Here's an example of an SSH config file that sets up port forwarding:

   ```
   Host example.com
     LocalForward 8080 localhost:80
   ```

   This configuration forwards traffic from port 8080 on the local machine to port 80 on the remote machine named "example.com".

3. Multiple hosts: You can define multiple hosts in the SSH config file. Here's an example:

   ```
   Host example.com
     User username
     Hostname example.com
     IdentityFile ~/.ssh/id_rsa

   Host example.net
     User username
     Hostname example.net
     IdentityFile ~/.ssh/id_rsa
   ```

   This configuration sets up two connections, one to "example.com" and another to "example.net". Each connection uses the same username and private key for authentication.

4. ProxyJump: The ProxyJump option allows you to connect to a remote host through an intermediate host. Here's an example:

   ```
   Host example.com
     User username
     Hostname example.com
     IdentityFile ~/.ssh/id_rsa
     ProxyJump intermediate.example.com
   ```

   This configuration sets up a connection to "example.com" through an intermediate server named "intermediate.example.com". The private key specified in the IdentityFile option is used for authentication.

   Overall, the SSH config file is a powerful tool for customizing SSH connections and automating the process of connecting to remote servers.

SSH Agent

The SSH agent holds your private keys in memory and automatically handles authentication requests. It's especially useful when working with multiple keys. Start the agent and add your private key:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_rsa
```

SSH Tunneling and Port Forwarding

SSH can securely forward traffic between local and remote ports:

- **Local Port Forwarding**: `ssh -L local-port:remote-host:remote-port user@remote-host`
- **Remote Port Forwarding**: `ssh -R remote-port:local-host:local-port user@remote-host`

Security Best Practices

1. Use strong key pairs (e.g., RSA with 4096 bits or ed25519).
2. Disable password authentication and use public key authentication instead.
3. Limit access to SSH by restricting the allowed IP addresses.
4. Use a non-standard SSH port to reduce scanning and brute-force attempts.
5. Enable multi-factor authentication (e.g., with Google Authenticator).
6. Regularly update the SSH client and server software.
7. Monitor and analyze SSH logs to detect and respond to security incidents.

There's a lot more to learn about SSH, but this guide should provide a solid foundation for a DevOps engineer to securely manage Linux environments.

▼ scp - Secure copy.

The scp (secure copy) command is a tool used to securely transfer files between hosts on a network. It uses the SSH (Secure Shell) protocol to provide encryption and authentication when transferring files. The basic syntax for the scp command is as follows:

```
scp [options] source_file destination_file
```

where `source_file` is the file that you want to transfer and `destination_file` is the location where you want to transfer it to.

Some of the commonly used options for the scp command are:

- `r` - Recursively copy entire directories.
- `P` - **Specifies the port number to use for the connection.**
- `p` - **Preserves the modification times, access times, and modes of the original file.**
- `v` - Verbose output mode.

Here are some examples of how to use the scp command:

1. **Transfer a file from a remote server to your local machine:**

   ```
   scp user@remote:/path/to/file /path/on/local/machine
   ```

   This command will copy the file located at `/path/to/file` on the remote server to `/path/on/local/machine` on your local machine.

2. **Transfer a file from your local machine to a remote server:**

   ```
   scp /path/on/local/machine user@remote:/path/to/destination
   ```

This command will copy the file located at `/path/on/local/machine` on your local machine to `/path/to/destination` on the remote server.

3. **Transfer an entire directory from a remote server to your local machine:**

   ```
   scp -r user@remote:/path/to/directory /path/on/local/machine
   ```

   This command will copy the entire directory located at `/path/to/directory` on the remote server to `/path/on/local/machine` on your local machine.

4. Transfer a file using a **different port number**:

   ```
   scp -P 2222 user@remote:/path/to/file /path/on/local/machine
   ```

   This command will use port number 2222 instead of the default port number (22) to transfer the file located at `/path/to/file` on the remote server to `/path/on/local/machine` on your local machine.

5. **Transfer a file and preserve its permissions:**

   ```
   scp -p user@remote:/path/to/file /path/on/local/machine
   ```

   This command will copy the file located at `/path/to/file` on the remote server to `/path/on/local/machine` on your local machine while preserving its permissions.

6. **Verbose output mode:**

   ```
   scp -v user@remote:/path/to/file /path/on/local/machine
   ```

   This command will copy the file located at `/path/to/file` on the remote server to `/path/on/local/machine` on your local machine while displaying verbose output about the transfer process.

   The possible output for each example depends on the specific files being transferred and the network conditions. However, the output will generally include information about the transfer process, such as the transfer speed, the amount of data transferred, and any errors or warnings that occur during the transfer. The verbose output mode (-v option) will provide more detailed information about the transfer process.

▼ ftp and sftp - File transfer protocol client and Secure File transfer protocol client.

In summary, both FTP and SFTP are used for transferring files over a network, but SFTP is a more secure and feature-rich protocol than FTP. SFTP is recommended for transferring sensitive data, while FTP may be used for non-sensitive data or in situations where SFTP is not available.

▼ curl - Transfer data from or to a server.

The "curl" command is a powerful tool for transferring data to or from a server using a variety of protocols, including HTTP, HTTPS, FTP, and more. It can be used to send requests to web services, download files, and perform other network-related tasks. Here are some variations of the "curl" command with examples:

1. Downloading a file:

   ```
   curl -O https://example.com/file.txt
   ```

   This command downloads the "file.txt" file from the server at "https://example.com" and saves it to the current directory on the local machine.

2. Uploading a file to a server:

   ```
   curl -F "file=@/path/to/file.txt" https://example.com/upload
   ```

   This command uploads the "file.txt" file from the local machine to the server at "https://example.com/upload" using the HTTP POST method and the "multipart/form-data" content type.

3. Setting a custom user-agent:

   ```
   curl -A "My User Agent" https://example.com
   ```

   This command sends a request to "https://example.com" with a custom user-agent header set to "My User Agent".

4. Following redirects:

   ```
   curl -L https://example.com
   ```

   This command follows any redirects returned by the server when accessing "https://example.com".

   The output of the "curl" command varies depending on the options used and the success or failure of the request. If the request is successful, the output will display the response returned by the server. If there are any errors, the output will provide information about the error, such as the HTTP status code or the reason for the failure. Overall, the "curl" command is a versatile tool for transferring data to or from a server using a wide variety of protocols and options.

5. Sending a GET request:

   ```
   curl https://example.com/api/users
   ```

   This command sends a GET request to the API endpoint at "https://example.com/api/users" and displays the response in the terminal window.

6. Sending a POST request:

   ```
   curl -X POST -d "username=johndoe&password=secret" https://example.com/api/login
   ```

   This command sends a POST request to the API endpoint at "https://example.com/api/login" with the parameters "username" and "password" set to "johndoe" and "secret", respectively.

▼ wget - Download files from the web.

The "wget" command is another powerful tool for downloading files from the internet. It's similar to the "curl" command, **but its primary focus is on downloading files** rather than sending requests to web services. Here are some variations of the "wget" command with examples:

Downloading a file:

```
wget https://example.com/file.txt
```

This command downloads the "file.txt" file from the server at "https://example.com" and saves it to the current directory on the local machine.

▼ Difference between Curl and wget

`curl` is a versatile tool that supports a wide range of protocols such as HTTP, FTP, SMTP, IMAP, POP3, and more. It can transfer data using various methods such as GET, POST, PUT, DELETE, and HEAD. `curl` can also be used for testing APIs, sending custom HTTP headers, and handling cookies and authentication.

`wget`, on the other hand, is a tool primarily used for downloading files from the internet. It supports HTTP, HTTPS, and FTP protocols and can download files recursively with options for limiting the depth and size of the download. `wget` can also pause and resume downloads, and it can be used to mirror entire websites for offline browsing.

Here are some of the key differences between `curl` and `wget`:

- `curl` supports more protocols than `wget`.
- `curl` can transfer data using various methods, while `wget` is primarily used for downloading files.
- `curl` is more versatile and can be used for testing APIs, handling cookies and authentication, and more.
- `wget` is better suited for downloading large files or mirroring entire websites.

In summary, `curl` is a more versatile tool for transferring data over the internet, while `wget` is a specialized tool for downloading files and mirroring websites.

▼ ping - test the reachability of a network host or IP address**

The ping command is a network utility tool **used to test the reachability of a network host or IP address by sending ICMP echo request packets** and receiving ICMP echo reply packets. It is one of the most commonly used network-based commands in Linux.

The basic syntax for the ping command is as follows:

```
ping [options] host
```

where `host` is the name or IP address of the network host that you want to test.

Here is an example of how to use the ping command:

```
ping google.com
```

This command will send ICMP echo request packets to the host `google.com` and wait for ICMP echo reply packets to be received. It will continue to send packets until interrupted by the user.

The possible output for this example is as follows:

```
PING google.com (142.250.68.206) 56(84) bytes of data.
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=1 ttl=117 time=8.01 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=2 ttl=117 time=7.95 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=3 ttl=117 time=8.13 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=4 ttl=117 time=8.11 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=5 ttl=117 time=8.12 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=6 ttl=117 time=8.18 ms
64 bytes from lga34s30-in-f14.1e100.net (142.250.68.206): icmp_seq=7 ttl=117 time=8.09 ms
^C
--- google.com ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6002ms
rtt min/avg/max/mdev = 7.947/8.076/8.181/0.094 ms
```

The initial line shows the host being pinged and its IP address. The following lines show the ICMP echo reply packets received from the host, including the size of the packet, the time it takes to receive a reply (in milliseconds), and the sequence number of the packet. The output also shows the number of packets transmitted and received, the percentage of packet loss, and the minimum, average, maximum, and standard deviation of the round-trip time (RTT) for the packets received.

In this example, the output shows that all 7 packets were successfully transmitted and received with an average RTT of 8.076 ms. This indicates that the host `google.com` is reachable and responsive on the network.

▼ nslookup - tool for querying the Domain Name System (DNS)**

The "nslookup" command is a powerful tool for querying the Domain Name System (DNS) to obtain information about domain names and IP addresses. It's a useful tool for troubleshooting DNS issues and verifying DNS records. Here are some examples of the "nslookup" command along with possible output:

Querying the IP address of a domain name:

```
  nslookup example.com
```

This command queries the DNS for the IP address of the "example.com" domain name. The possible output could look like this:

```
Server:  resolver1.opendns.com
Address:  208.67.222.222

Non-authoritative answer:
Name:    example.com
Address:  93.184.216.34
```

In this example, the DNS server used to resolve the query is "resolver1.opendns.com" with IP address "208.67.222.222". The non-authoritative answer shows that the IP address of "example.com" is "93.184.216.34".

▼ mail - sending and receiving email messages from the command line

The mail command is a powerful tool for sending and receiving email messages from the command line in Unix and Linux systems. Here is an exhaustive explanation of the mail command, including its variations, examples, and possible output for each example:

Basic Syntax:

```
mail [options] [recipient1] [recipient2] ...
```

Options:

- `s` : specify the subject of the email
- `a` : attach a file to the email
- `c` : send a copy (cc) of the email to additional recipients
- `b` : send a blind copy (bcc) of the email to additional recipients
- `q` : quiet mode, suppress notifications

Examples:

1. Send an email message:

   ```
   mail user@example.com
   ```

   This will open the mail editor, where you can enter the message body. Press Ctrl+D to send the message.

2. Send an email message with a subject:

   ```
   mail -s "Test message" user@example.com
   ```

   This will open the mail editor with the specified subject. Enter the message body and press Ctrl+D to send the message.

3. Send an email message with an attachment:

   ```
   mail -s "Test message with attachment" -a /path/to/file.txt user@example.com
   ```

   This will attach the specified file to the email message. Enter the message body and press Ctrl+D to send the message.

4. Send an email message to multiple recipients:

   ```
   mail user1@example.com user2@example.com
   ```

   This will send the email message to both recipients. Enter the message body and press Ctrl+D to send the message.

5. Send a copy of the email message to additional recipients:

   ```
   mail -s "Test message with cc" -c user2@example.com user1@example.com
   ```

   This will send the email message to the primary recipient and cc a copy to the additional recipient. Enter the message body and press Ctrl+D to send the message.

6. Send a blind copy of the email message to additional recipients:

   ```
   mail -s "Test message with bcc" -b user2@example.com user1@example.com
   ```

   This will send the email message to the primary recipient and bcc a copy to the additional recipient. Enter the message body and press Ctrl+D to send the message.

7. Send an email message without notifications:

```
mail -q -s "Test message without notifications" user@example.com
```

This will send the email message without generating any notifications. Enter the message body and press Ctrl+D to send the message.

Possible Output:

The output of the mail command depends on the specific example and options used. In general, the command will display messages indicating whether the email was sent successfully or encountered any errors. If the mail editor is opened, the command will display prompts to enter the message body and send the message. If an attachment is included, the command may display additional messages indicating the status of the attachment. In some cases, the command may also generate notifications or error messages that are sent to the sender or other recipients.

▼ Compression and Archiving

▼ gzip - Compress files.

The "gzip" command is a popular tool for **compressing and decompressing files using the gzip compression algorithm**. It's widely used for reducing the size of files and improving transfer speeds over the internet. Here are some variations of the "gzip" command with examples and possible output for each example:

1. **Compressing a file:**

   ```
   gzip file.txt
   ```

   This command compresses the "file.txt" file using the gzip compression algorithm and saves the compressed file as "file.txt.gz" in the same directory. The possible output could look like this:

   ```
   file.txt.gz
   ```

   In this example, the compressed file "file.txt.gz" is created in the same directory.

2. **Compressing multiple files:**

   ```
   gzip file1.txt file2.txt
   ```

   This command compresses the "file1.txt" and "file2.txt" files using the gzip compression algorithm and saves the compressed files as "file1.txt.gz" and "file2.txt.gz" in the same directory. The possible output could look like this:

   ```
   file1.txt.gz  file2.txt.gz
   ```

   In this example, the compressed files "file1.txt.gz" and "file2.txt.gz" are created in the same directory.

3. **Compressing a file and keeping the original file:**

   ```
   gzip -k file.txt
   ```

   This command compresses the "file.txt" file using the gzip compression algorithm and saves the compressed file as "file.txt.gz" in the same directory. The original file "file.txt" is kept unchanged. The possible output could look like this:

   ```
   file.txt.gz
   ```

   In this example, the compressed file "file.txt.gz" is created in the same directory, and the original file "file.txt" is kept unchanged.

4. **Compressing a file and specifying the compression level:**

   ```
   gzip -9 file.txt
   ```

This command compresses the "file.txt" file using the gzip compression algorithm with the **highest compression level (-9)** and saves the compressed file as "file.txt.gz" in the same directory. The possible output could look like this:

```
file.txt.gz
```

In this example, the compressed file "file.txt.gz" is created in the same directory with the highest compression level.

5. **Decompressing a file:**

```
gzip -d file.txt.gz
```

This command decompresses the "file.txt.gz" file using the gzip compression algorithm and saves the decompressed file as "file.txt" in the same directory. The possible output could look like this:

```
file.txt
```

In this example, the decompressed file "file.txt" is created in the same directory.

6. **Displaying the compression ratio:**

```
gzip -l file.txt.gz
```

This command displays the compression ratio of the "file.txt.gz" file. The possible output could look like this:

```
         compressed        uncompressed  ratio uncompressed_name
                  43                   61.5%            file.txt
```

In this example, the compressed file size is 43 bytes, the uncompressed size is 61.5%, and the file name is "file.txt".

Overall, the "gzip" command is a useful tool for compressing and decompressing files using the gzip compression algorithm. The output of the "gzip" command provides valuable information about the compression ratio and the file names of the compressed files.

▼ zip - Compress files into a zip archive.

The zip command is a popular utility in Linux used to compress one or more files into a single archive file. It supports a variety of compression algorithms and provides various options to customize the compression process.

The basic syntax for the zip command is as follows:

```
zip [options] archive_name file1 file2 ...
```

where `archive_name` is the name of the archive file to create or manipulate, and `file1`, `file2`, etc. are the files or directories to include in the archive.

Here are some commonly used options for the `zip` command:

- `r`: recursively include all files and directories in the archive
- `q`: quiet mode, suppress output messages during compression
- `v`: verbose mode, show the progress and details of the compression operation
- `j`: store only the file data, without including directory information
- `9`: use maximum compression level
- `u`: update existing archive with new files
- `d`: delete files from the archive
- `T`: test the integrity of the archive
- `m`: move the original files to the archive after compression
- `c`: ~~create a new archive file~~

- `x`: exclude specific files or directories from the archive
- `z`: compress the archive using gzip compression
- `0`: store files in the archive without compression
- `1` to `8`: specify compression level, where `1` is fastest and `8` is slowest but most effective

Here are some examples of how to use the zip command:

1. Create a zip archive:

```
zip archive.zip file1 file2 file3
```

This command will create a new archive file named `archive.zip` and include the files `file1`, `file2`, and `file3`.

The possible output for this example is as follows:

```
  adding: file1 (stored 0%)
  adding: file2 (stored 0%)
  adding: file3 (stored 0%)
```

This output shows that the files were added to the archive and the compression ratio (in this case, the files were stored with no compression).

2. Extract files from a zip archive:

```
unzip archive.zip
```

This command will extract the files from the archive file `archive.zip` and write them to the current directory.

The possible output for this example is as follows:

```
Archive:  archive.zip
  inflating: file1
  inflating: file2
  inflating: file3
```

This output shows that the files were extracted from the archive.

3. Compress a zip archive using a specific compression level:

```
zip -9 archive.zip file1 file2 file3
```

This command will create a new archive file named `archive.zip` and include the files `file1`, `file2`, and `file3`, compressing them using the maximum compression level (`-9`).

The possible output for this example is the same as Example 1, which shows that the files were added to the archive.

4. Extract a specific file from a zip archive:

```
unzip archive.zip file1
```

This command will extract only the file `file1` from the archive file `archive.zip` and write it to the current directory.

The possible output for this example is the same as Example 2, which shows that the file was extracted from the archive.

There are many other options and variations of the zip command, including the ability to add or remove files from an existing archive, password-protect an archive, and compress files using different compression algorithms. By mastering the zip command, you can compress and manage files efficiently and effectively on your Linux system.

gzip vs zip command in linux

The "gzip" command uses the gzip compression algorithm, which is a fast and efficient algorithm that produces compressed files with a ".gz" extension. It's commonly used for compressing **large files and for transferring files over the internet.**

The "zip" command, on the other hand, uses the zip compression algorithm, which is a slower but more powerful algorithm that produces compressed files with a ".zip" extension. It's commonly used for creating archives of **multiple files and for compressing files with encryption.**

tar - tape Archive files.

The tar (tape archive) command is used to create and manipulate archive files in Linux. It allows you to combine multiple files and directories into a single **archive** file, which can then be **compressed** and transferred to another machine or stored for backup purposes.

The basic syntax for the tar command is as follows:

```
tar [options] archive_name file1 file2 ...
```

where `archive_name` is the name of the archive file to create or manipulate, and `file1`, `file2`, etc. are the files or directories to include in the archive.

Here are the commonly used options for the `tar` command:

- `c`: **create** a new archive file
- `x`: **extract** files from an archive file
- `f`: specify the **name** of the archive file
- `v`: **verbose** mode, show the progress and details of the archive operation
- `z`: **compress** the archive file using gzip compression
- `j`: compress the archive file using bzip2 compression
- `p`: **preserve** permissions and file attributes during extraction
- `r`: append files to an existing archive file
- `u`: update files in an existing archive file
- `A`: append files to an existing archive, but only include files that are newer than the current archive contents
- `B`: don't back up files that are newer than the current archive contents
- `C`: change to a specific directory before performing the archive operation
- `-exclude`: exclude specific files or directories from the archive
- `-remove-files`: remove files after adding them to the archive
- `-directory`: specify the directory where the archive file should be extracted to
- `-overwrite`: overwrite existing files during extraction without prompting for confirmation
- `-wildcards`: use wildcards to specify files and directories to include in the archive

Here are some examples of how to use the tar command:

1. Create a tar archive:

   ```
   tar -cvf archive.tar file1 file2 file3
   ```

   This command will create a new archive file named `archive.tar` and include the files `file1`, `file2`, and `file3`. The `-c` option specifies that a new archive should be created, and the `-v` option specifies that verbose output should be displayed.

   The possible output for this example is as follows:

   ```
   file1
   file2
   file3
   ```

   This output shows the names of the files that were added to the archive.

2. Extract files from a tar archive:

```
tar -xvf archive.tar
```

This command will extract the files from the archive file `archive.tar` and write them to the current directory. The `-x` option specifies that files should be extracted from the archive, and the `-v` option specifies that verbose output should be displayed.

The possible output for this example is the same as the output from the previous example, which shows the names of the files that were extracted.

3. Compress a tar archive:

```
tar -czvf archive.tar.gz file1 file2 file3
```

This command will create a new compressed archive file named `archive.tar.gz` and include the files `file1`, `file2`, and `file3`. The `-c` option specifies that a new archive should be created, the `-z` option specifies that the archive should be compressed using gzip, and the `-v` option specifies that verbose output should be displayed.

The possible output for this example is the same as the output from Example 1, which shows the names of the files that were added to the archive.

4. Extract a compressed tar archive:

```
tar -xzvf archive.tar.gz
```

This command will extract the files from the compressed archive file `archive.tar.gz` and write them to the current directory. The `-x` option specifies that files should be extracted from the archive, the `-z` option specifies that the archive is compressed using gzip, and the `-v` option specifies that verbose output should be displayed.

The possible output for this example is the same as the output from Examples 1 and 2, which shows the names of the files that were extracted.

There are many other options and variations of the tar command, including the ability to specify compression algorithms other than gzip, exclude files or directories from the archive, and specify the location of the archive file. By mastering the tar command, you can create and manage archives efficiently and effectively on your Linux system.

▼ Difference between gzip, zip and tar commands

`gzip`, `zip`, and `tar` are all command-line tools used for file compression and archiving, but they have different functionalities.

`gzip` is a compression tool that works by compressing a **single file**. It replaces the original file with a compressed version that has a `.gz` extension. Its primary use is to compress large files to save disk space or to reduce the time it takes to transfer files over a network.

`zip` is a compression and archiving tool that can compress **multiple files and directories** into a single archive file with a `.zip` extension. `zip` is commonly used to **create backups, archive files for distribution**, or to compress multiple files into a single archive for easier transfer.

`tar` is an archiving tool that can be used to create a single archive file from multiple files and directories. Unlike `zip`, `tar` **does not compress files by default,** but it can be used in conjunction with other compression tools, such as `gzip` or `bzip2`, to create a compressed archive. `tar` is commonly used for creating backups, distributing files, or transferring files over a network.

In summary, `gzip` is used for compressing a single file, `zip` is used for compressing and archiving multiple files and directories, and `tar` is used for archiving multiple files and directories, which can then be compressed using a separate compression tool.

▼ System Administration

▼ sudo - Execute a command as a superuser.

The "sudo" command in Linux allows a user to run a command **with the privileges of another user, typically the root user**. This is useful for performing administrative tasks that require elevated privileges, such as installing software or modifying system files.

Here's how the "sudo" command works:

1. When you run a command with "sudo", it checks the "/etc/sudoers" file to determine if your user account is authorized to use "sudo".
2. If your user account is authorized, you'll be prompted to enter your user password. This is to confirm your identity before executing the command with elevated privileges.

3. If your password is correct, the command will be executed with the privileges of the user specified in the "/etc/sudoers" file. By default, this is the root user.
4. If the command requires additional arguments or options, you can specify them after the command name, like in the example above.

Overall, the "sudo" command is a useful tool for running commands with elevated privileges in Linux, and it helps to improve security by limiting the use of the root account.

▼ **How to give sudo privileges to a user**

To give sudo privileges to a user in Linux, follow these steps:

1. Log in to your Linux system as a user with root privileges.
2. Open the "/etc/sudoers" file for editing using a text editor like "nano" or "vim":

```
sudo visudo
```

▼ visudo - Used to edit the sudoers file, which controls who can use the sudo command.

The "visudo" command in Linux is used to edit the sudoers file, which controls the permissions and access of users and groups to perform administrative tasks using the "sudo" command.

The "visudo" command is used instead of directly editing the sudoers file with a text editor, as it provides a more secure method of editing the file and ensures that the syntax and permissions are valid before saving the changes.

Here is an example of how to use the "visudo" command:

```
sudo visudo
```

When you run this command, it will open the sudoers file in the default text editor specified in the "EDITOR" environment variable (usually vi or vim). You can then make changes to the file using the appropriate syntax and permissions for the tasks and users you want to configure.

3. In the "/etc/sudoers" file, locate the line that reads:

```
root    ALL=(ALL:ALL) ALL
```

4. Below the "root" line, add a new line that specifies the username of the user you want to give sudo privileges to, followed by the word "ALL". For example:

```
username    ALL=(ALL:ALL) ALL
```

Replace "username" with the actual username of the user you want to give sudo privileges to.

1. Save the changes to the "/etc/sudoers" file and exit the text editor.
2. Verify that the user has sudo privileges by logging out and logging back in as that user, then running a command with "sudo". For example:

```
sudo apt-get update
```

If the user is prompted for their password and the command executes successfully, the user has sudo privileges.

Note: It's important to be careful when editing the "/etc/sudoers" file, as incorrect syntax can cause the file to become unusable. It's recommended to use the "visudo" command to edit the file, as it performs syntax checking and prevents multiple users from editing the file simultaneously.

▼ chown - Change file owner and group.

The "chown" command in Linux is used to change the ownership of files and directories. Here are some variations of the "chown" command with examples and possible output for each example:

1. **Changing the ownership of a file:**

   ```
   sudo chown user:group filename
   ```

   This command changes the ownership of the "filename" file to the "user" user and the "group" group. The possible output could look like this:

   ```
   sudo chown john:users file.txt
   ```

   In this example, the ownership of the "file.txt" file is changed to the "john" user and the "users" group, and the output does not display any confirmation or error messages.

2. Changing the ownership of a directory and its contents:

   ```
   sudo chown -R user:group directory
   ```

   This command changes the ownership of the "directory" directory **and all of its contents** to the "user" user and the "group" group. The "-R" option specifies that the ownership change should be applied recursively to all subdirectories and files within the directory. The possible output could look like this:

   ```
   sudo chown -R john:users /var/www/html
   ```

   In this example, the ownership of the "/var/www/html" directory and all of its contents is changed to the "john" user and the "users" group, and the output does not display any confirmation or error messages.

3. **Changing the ownership of a symbolic link:**

   ```
   sudo chown user:group symlink
   ```

   This command changes the ownership of the symbolic link "symlink" itself, not the file or directory that it points to. The possible output could look like this:

   ```
   sudo chown john:users link.txt
   ```

   In this example, the ownership of the "link.txt" symbolic link is changed to the "john" user and the "users" group, and the output does not display any confirmation or error messages.

4. **Changing the ownership of multiple files or directories:**

   ```
   sudo chown user:group file1 file2 directory1
   ```

   This command changes the ownership of the "file1" file, "file2" file, and "directory1" directory to the "user" user and the "group" group. The possible output could look like this:

   ```
   sudo chown john:users file1.txt file2.txt /var/www/html
   ```

   In this example, the ownership of the "file1.txt" and "file2.txt" files, and the "/var/www/html" directory is changed to the "john" user and the "users" group, and the output does not display any confirmation or error messages.

   It's important to use the "chown" command carefully and follow best practices for ownership and permissions management, as incorrect ownership or permissions can potentially grant unauthorized access to system resources. Always verify the ownership and permissions of files and directories before and after making changes, and test the changes on a non-production system or in a test environment before applying them to a production system.

▼ chmod - Change file permissions.

The chmod command is used to change the permissions of files and directories in Linux. It is an essential command for managing file and directory permissions and is used to control which users and groups can read, write, and execute files and directories.

The basic syntax for the chmod command is as follows:

```
chmod [options] mode file
```

where `mode` is the new permissions to set for the file or directory, and `file` is the name of the file or directory to modify.

The `mode` parameter can be specified in several ways:

1. **Symbolic notation** - This method uses symbols to represent the permissions that are being set. The symbols include `u` for user, `g` for group, `o` for others, and `a` for all. The permissions are represented by `r` for read, `w` for write, and `x` for execute. For example, the command `chmod u+r file.txt` adds read permission for the user.
2. **Numeric notation** - This method uses a three-digit number to represent the permissions that are being set. The digits represent the user, group, and others, respectively, and the numbers 4, 2, and 1 represent read, write, and execute permissions. For example, the command `chmod 644 file.txt` sets read and write permission for the user and read permission for the group and others.

Here are some examples of how to use the chmod command:

1. Add read, write, and execute permissions for the user and group:

   ```
   chmod ug+rwx file.txt
   ```

   This command will add read, write, and execute permissions for the user and group on the `file.txt` file.

2. Remove write permission for others:

   ```
   chmod o-w file.txt
   ```

   This command will remove write permission for others on the `file.txt` file.

3. Change permissions using numeric notation:

   ```
   chmod 644 file.txt
   ```

   This command will set read and write permission **for the user** and read permission **for the group** and others on the `file.txt` file.

   The possible output for this example is the same as the previous examples, which indicates that the permissions for the file were changed successfully.

   Here,

   - To give user full permission i.e read + write + execute (or) rwx = 4+2+1 = 7
   - To give no permission = 0
   - To give only write and execute = 2+1 = 3
   - To give user, group and others, full permissions = 777

   There are many other options and variations of the chmod command, including the ability to set permissions recursively for a directory and to view the current permissions of a file or directory. By mastering the chmod command, you can manage file and directory permissions efficiently and effectively on your Linux system.

▼ umask ("user file creation mask") - Set the default file permissions.

The "umask" command is a shell command in Linux and Unix-based operating systems that sets the default file creation permissions for new files and directories. It stands for "user mask" and is represented by a three-digit octal number that specifies the permissions to be removed from the default settings.

The `umask` command takes a single argument, which is a numeric value representing the mask. The mask is a three-digit octal number that specifies the permissions to be masked. Each digit represents the permissions for a different group: the owner, the group, and others. The mask is subtracted from the default permissions to determine the actual permissions for newly created files.

> 💡 The default file and directory permissions for newly created files and directories in Linux are:
>
> - For files: `644` (read-write for owner, read-only for group and others)
> - For directories: `755` (read-write-execute for owner, read-execute for group and others)

> 💡 The default `umask` value in Linux is usually `022`. This means that write permission is removed from the group and others permissions.

> 💡 **When umask value is 0,** For files, the default permissions are `666` (read-write for owner, group, and others), and for directories, the default permissions are `777` (read-write-execute for owner, group, and others).

For example, if the default permissions for newly created files are 644 i.e `rw-r--r--` (read-write for owner, read-only for group and others), and the `umask` value is `022`, the actual permissions for new files will be `rw-r--r--` minus `022`, which results in `rw-r--r--` for owner, and `r--r--r--` for group and others.

Here is an explanation of the "umask" command, its variations, and examples:

1. Basic Usage:The basic syntax of the "umask" command is:

```
umask [mask]
```

The mask is a three-digit octal number that specifies the permissions to be removed from the default settings. by default**, a umask of 022 will remove write permission from group and others, leaving the default permissions as 755 (rwxr-xr-x) for directories and 644 (rw-r--r--) for files.**

1. Variations: There are two variations of the "umask" command:
   - System-wide umask: This sets the default file creation permissions for all users on the system. It is usually set in the "/etc/profile" file or the "/etc/login.defs" file.
   - User-specific umask: This sets the default file creation permissions for a specific user. It is usually set in the user's "/.bashrc" or "/.bash_profile" file.
2. Examples: Here are some examples of how to use the "umask" command:
   - To set the default file creation permissions to 755 (rwxr-xr-x), use the following command:

```
umask 022
#this is the default permission set already
```

   - To set the default file creation permissions to 777 (rwxrwxrwx), use the following command:

```
umask 000
```

   - To set the default file creation permissions to 644 (rw-r--r--), use the following command:

```
umask 022
```

   - To set the default file creation permissions to 600 (rw-------) for a specific user, add the following line to the user's "~~/.bashrc" or "~~/.bash_profile" file:

```
umask 077
```

3. For example, if you set the umask to 022, the default file creation permissions will be 755 (rwxr-xr-x). To verify this, you can create a new directory using the following command:

```
mkdir testdir
```

Then, you can use the "ls -l" command to view the permissions of the new directory:

```
ls -l testdir
```

The output should show the permissions as:

```
drwxr-xr-x 2 user group 4096 Mar 30 10:00 testdir
```

Note that the permissions are set to rwxr-xr-x, which is the default permissions with the umask set to 022.

▼ crontab - Schedule and Manage cron jobs.

The crontab command is used to schedule periodic tasks or jobs to run automatically at specific times on a Unix or Linux system. It is a powerful tool that can help automate routine tasks and reduce the need for manual intervention.

Crontab.guru - The cron schedule expression editor

An easy to use editor for crontab schedules.

https://crontab.guru/

Install the cron package by running the following command:

```
sudo yum install cronie
```

The basic syntax for the crontab command is as follows:

```
crontab [options] file
```

where `file` is the name of the file containing the cron jobs to be scheduled.

Here are some examples of how to use the crontab command:

1. List the current user's crontab:

```
crontab -l
```

This command will display the current user's crontab, **which contains a list of scheduled cron jobs.**

The possible output for this example is as follows:

```
* * * * * /path/to/command
```

This output indicates that a command will run every minute.

Syntax:

```
*     *     *     *     *

┬     ┬     ┬     ┬     ┬
│     │     │     │     │
│     │     │     │     │
│     │     │     │     └───── day of the week (0-6, Sunday is 0 or 7)
│     │     │     └────────── month (1-12)
```

```
|     |     └────────────── day of the month (1-31)
|     └─────────────────── hour (0-23)
└──────────────────────── minute (0-59)
```

💡 **"Maya Hopes David Moves West”**

2. Edit the current user's crontab:

   ```
   crontab -e
   ```

   This command will open the current user's crontab file in a text editor, allowing the user to add or modify cron jobs.

   The possible output for this example is the crontab file itself, which will be displayed in the text editor.

3. Schedule a cron job to run every day at midnight:

   ```
   0 0 * * * /path/to/command
   ```

   This command will schedule a cron job to run at midnight every day.

   The possible output for this example is the same as the output for the first example, which indicates that a command will run every minute.

4. Schedule a cron job to run every Monday at 8:30 AM:

   ```
   30 8 * * 1 /path/to/command
   ```

   This command will schedule a cron job to run every Monday at 8:30 AM.

   The possible output for this example is the same as the output for the first example, which indicates that a command will run every minute.

   There are many other options and variations of the crontab command, including the ability to specify the user whose crontab is being edited, to remove cron jobs, and to view the system-wide crontab file. By mastering the crontab command, you can automate routine tasks and schedule jobs to run automatically at specific times, increasing your productivity and efficiency.

▼ **How to make a cronjob that records date in a file every minute in linux**

To create a cron job that records the current date and time to a file every minute, you can use the following steps:

1. Open your crontab file using the command `crontab -e`.
2. Add the following line to the file:

   applescript

   Copy

   ~~~
   * * * * * date >> /path/to/file.txt
   ```

   Replace `/path/to/file.txt` with the actual path and [file name](poe://www.poe.com/_api/key_phrase?phrase=file%
   20name&prompt=Tell%20me%20more%20about%20file%20name.) where you want to save the date and time information.

   # To define the time you can provide concrete values for
   # minute (m), hour (h), day of month (dom), month (mon),
   # and day of week (dow) or use '*' in these fields (for 'any').
   ~~~

3. Save the file and exit the text editor.

This cron job will run every minute (denoted by the five asterisks) and execute the `date` command, which will output the current date and time. The `>>` operator will append the output to the specified file.

▼ **what are the top 5 uses of crontab command**

The crontab command is a powerful tool for scheduling periodic tasks or jobs to run automatically on a Unix or Linux system. Here are the top 5 uses of the crontab command:

1. **Regular backups** - You can use the crontab command to schedule regular backups of important files or databases. For example, you can schedule a cron job to run a backup script every day, week, or month, depending on your backup requirements.

   ▼ **schedule a cron job to run a backup script every day. explain everything in great detail and step by step**

   Sure, here is a step-by-step guide on how to schedule a cron job to run a backup script every day:

   1. Create a backup script: First, create a shell script that performs the backup operation. The script should include commands to copy or archive the files or directories that you want to back up. For example, you can create a script named `backup.sh` that copies the contents of the `/var/www/html` directory to a backup directory:

      ```
      #!/bin/bash
      mkdir -p /backup/html
      cp -r /var/www/html/* /backup/html
      ```

      Save the script to a location of your choice, such as `/home/user/scripts/backup.sh`.

   2. Make the script executable: Next, make the script executable by running the following command:

      ```
      chmod +x /home/user/scripts/backup.sh
      ```

      This will allow the script to be executed as a program.

   3. Edit the crontab file: Use the following command to edit the crontab file for the current user:

      ```
      crontab -e
      ```

      This will open the crontab file in the default text editor.

   4. Schedule the cron job: In the crontab file, add the following line to schedule the backup script to run every day at midnight:

      ```
      0 0 * * * /home/user/scripts/backup.sh
      ```

      This line specifies that the script should run at minute 0 and hour 0 (midnight) every day of the month, every month, and every day of the week.

   5. Save and exit the crontab file: Save the changes to the crontab file and exit the text editor.

   6. Verify the cron job: Use the following command to verify that the cron job has been scheduled:

      ```
      crontab -l
      ```

      This will display the contents of the crontab file, including the newly scheduled backup job.

   7. Monitor the backup process: Wait until midnight and check that the backup script has run successfully. You can check the backup directory to see if the backup files have been created.

      That's it! Now you have scheduled a cron job to run a backup script every day, automatically backing up your important files or directories on a regular basis.

2. **System maintenance** - You can use the crontab command to schedule system maintenance tasks, such as cleaning up log files, updating software packages, or rebooting the system at regular intervals.

   ▼ **schedule system maintenance tasks, such as cleaning up log files, updating software packages, or rebooting the system at regular intervals. explain everything in great detail and step by step**

   Sure, here is a step-by-step guide on how to use the crontab command to schedule system maintenance tasks:

   1. Create a maintenance script: First, create a shell script that performs the maintenance tasks that you want to automate. For example, you can create a script named `maintenance.sh` that cleans up old log files, updates

      software packages, and reboots the system:

      ```
      #!/bin/bash
      # Clean up log files older than 7 days
      find /var/log -type f -mtime +7 -delete

      # Update software packages
      apt-get update -y
      apt-get upgrade -y

      # Reboot the system
      reboot
      ```

      Save the script to a location of your choice, such as `/home/user/scripts/maintenance.sh`.

   2. Make the script executable: Next, make the script executable by running the following command:

      ```
      chmod +x /home/user/scripts/maintenance.sh
      ```

      This will allow the script to be executed as a program.

   3. Edit the crontab file: Use the following command to edit the crontab file for the current user:

      ```
      crontab -e
      ```

      This will open the crontab file in the default text editor.

   4. Schedule the cron job: In the crontab file, add the following line to schedule the maintenance script to run at regular intervals:

      ```
      0 0 * * * /home/user/scripts/maintenance.sh
      ```

      This line specifies that the maintenance script should run at midnight every day.

   5. Save and exit the crontab file: Save the changes to the crontab file and exit the text editor.

   6. Verify the cron job: Use the following command to verify that the cron job has been scheduled:

      ```
      crontab -l
      ```

      This will display the contents of the crontab file, including the newly scheduled maintenance job.

   7. Monitor the maintenance process: Wait until midnight and check that the maintenance script has run successfully. You can check the log files to make sure that old logs have been deleted, and check the system status to make sure that software packages have been updated and the system has been rebooted.

      That's it! Now you have scheduled a cron job to automate system maintenance tasks, reducing the need for manual intervention and improving system efficiency. Note that the exact maintenance tasks and schedule may vary depending on your system requirements and preferences.

3. **Automated testing** - You can use the crontab command to schedule automated testing of software applications, scripts, or websites. For example, you can schedule a cron job to run a test suite every night, to ensure that your software is working as expected.

   ▼ **schedule a cron job to run a test suite every night, to ensure that your software is working as expected.**

   Sure, here is a step-by-step guide on how to schedule a cron job to run a test suite every night:

   1. Create a test script: First, create a shell script that runs the test suite for your software application. The script should include commands to run the tests and capture the results. For example, you can create a script named `test.sh` that runs the test suite using the `pytest` testing framework:

      ```
      #!/bin/bash
      cd /path/to/your/application
      ```

```
pytest tests/
```

Save the script to a location of your choice, such as `/home/user/scripts/test.sh`.

same as previous case

4. **Data processing** - You can use the crontab command to schedule data processing tasks, such as data import/export, data analysis, or data transformation. For example, you can schedule a cron job to run a script that processes data from a database or file every hour, day, or week.
5. **Email notifications** - You can use the crontab command to schedule email notifications for important events, such as system errors, disk space warnings, or security alerts. For example, you can schedule a cron job to run a script that sends an email notification to the system administrator whenever a critical event occurs.

▼ schedule a cron job to run a script that sends an email notification to the system administrator when system errors

Sure, here is a step-by-step guide on how to schedule a cron job to send an email notification to the system administrator when system errors occur:

1. Create an email script: First, create a shell script that sends an email notification when system errors occur. The script should include commands to check for errors and send an email if errors are found. For example, you can create a script named `error_check.sh` that checks the system logs for errors and sends an email notification if errors are found:

```
#!/bin/bash
LOGFILE=/var/log/syslog

# Check for error messages in the system log
if grep -i "error" $LOGFILE >/dev/null; then
    # Send an email notification to the system administrator
    echo "Error found in system log file" | mail -s "Error notification" admin@example.com
fi
```

Save the script to a location of your choice, such as `/home/user/scripts/error_check.sh`.

▼ env - Display environment variables.

The env command is used to **display the current environment variables** or run a command with modified environment variables. Here is an explanation of the env command, including examples and possible output:

Basic Syntax:

```
env [OPTION]... [-] [NAME=VALUE]... [COMMAND [ARG]...]
```

Options:

- `i` : start with an empty environment
- `u` : unset the specified environment variable
- `P` : search for the specified command in the PATH environment variable only
- `S` : set the specified secure environment variable

Examples:

1. Display the current environment variables:

```
env
```

This will display a list of the current environment variables, including variables such as `HOME`, `PATH`, and `USER`.

2. Display the value of a specific environment variable:

```
env HOME
or
echo $HOME
```

This will display the value of the `HOME` environment variable.

3. Set a new environment variable:

```
env FOO=bar
```

This will set a new environment variable named `FOO` with the value `bar`.

4. Run a command with modified environment variables:

```
env PATH=/usr/local/bin:/usr/bin:/bin ls
```

This will run the `ls` command with a modified `PATH` environment variable, which specifies the directories to search for executables. The modified `PATH` variable includes the directories `/usr/local/bin`, `/usr/bin`, and `/bin`.

▼ **what are environment variables explain like I am ten**

Environment variables are like secret codes that the computer uses to remember important information about itself and its users.

For example, imagine you have a secret code named `FAVORITE_COLOR` that you tell your computer. Whenever your computer sees the code `FAVORITE_COLOR`, it knows what your favorite color is and can use that information to make decisions.

Environment variables are similar to this secret code. They are special codes that the computer uses to remember important information, such as the username, the computer's name, or the location of certain programs.

These environment variables can be used by programs or scripts to help them work better. For example, a program might use the `HOME` environment variable to find the user's home directory, or the `PATH` environment variable to find the location of certain programs.

Overall, environment variables are like secret codes that help the computer remember important information and make decisions based on that information. They are used by programs and scripts to work better and make the computer run more smoothly.

Examples:

Here are some examples of how environment variables can be used in Unix and Linux systems:

1. Configuration:

```
JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
```

In this example, the `JAVA_HOME` environment variable is set to the location of the Java Development Kit (JDK) on the system. This variable is used by many Java applications to locate the JDK and run Java code.

2. System information:

```
HOSTNAME=mycomputer.local
```

In this example, the `HOSTNAME` environment variable is set to the name of the computer on the local network. This variable can be useful for troubleshooting network issues or for identifying the computer in a multi-computer environment.

3. Security:

```
API_KEY=abc123xyz
```

In this example, the `API_KEY` environment variable is set to a secret key that is used to authenticate with an API. **By storing the key in an environment variable, the key can be easily accessed by scripts and programs without being exposed in plain text.**

▼ **explain more about this with example in terraform tool : By storing the key in an environment variable, the key can be easily accessed by scripts and programs without being exposed in plain text.**

In Terraform, environment variables are often used to securely store sensitive information, such as API keys or access tokens, that are needed by Terraform scripts to interact with cloud providers or other services.

For example, let's say you are using Terraform to provision infrastructure on a cloud provider such as AWS. To authenticate with AWS, you need to provide an access key and a secret key. However, storing these keys in plain text in your Terraform script or configuration files is not secure, as anyone who has access to the files can see the keys.

To securely store the access and secret keys, you can use environment variables. You can set the access and secret keys as environment variables on your local machine or on the server where Terraform is running. For example:

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Once the environment variables are set, you can reference them in your Terraform scripts using the `${var.VARIABLE_NAME}` syntax. For example:

```
provider "aws" {
  access_key = "${var.AWS_ACCESS_KEY_ID}"
  secret_key = "${var.AWS_SECRET_ACCESS_KEY}"
  region     = "us-west-2"
}
```

By using environment variables to store the access and secret keys, you can easily access the keys in your Terraform scripts without exposing them in plain text. This helps to keep your sensitive information secure and reduces the risk of unauthorized access to your cloud resources.

Overall, using environment variables is a best practice for storing sensitive information in Terraform and other tools. It allows you to easily and securely access the information you need without exposing it in plain text in your scripts or configuration files.

▼ **explain more about this with example in ansible tool : By storing the key in an environment variable, the key can be easily accessed by scripts and programs without being exposed in plain text.**

In Ansible, environment variables can also be used to securely store sensitive information that is needed by Ansible playbooks to interact with systems or cloud providers.

For example, let's say you are using Ansible to deploy and configure a web application on multiple servers. To authenticate with the database server, you need to provide a username and password. However, storing these credentials in plain text in your Ansible playbook or inventory file is not secure, as anyone who has access to the files can see the credentials.

To securely store the database credentials, you can use environment variables. You can set the username and password as environment variables on your local machine or on the Ansible control node. For example:

```
export DB_USERNAME=myusername
export DB_PASSWORD=mypassword
```

Once the environment variables are set, you can reference them in your Ansible playbooks using the `ansible_env` dictionary. For example:

```
- name: Configure database connection
  vars:
    db_username: "{{ ansible_env.DB_USERNAME }}"
    db_password: "{{ ansible_env.DB_PASSWORD }}"
  mysql_user:
    name: "{{ db_username }}"
    password: "{{ db_password }}"
    login_unix_socket: /var/run/mysqld/mysqld.sock
```

By using environment variables to store the database credentials, you can easily access the credentials in your Ansible playbooks without exposing them in plain text. This helps to keep your sensitive information secure and reduces the risk of unauthorized access to your systems or cloud resources.

Overall, using environment variables is a best practice for storing sensitive information in Ansible and other tools. It allows you to easily and securely access the information you need without exposing it in plain text in your playbooks or inventory files.

4. User information:

```
HOME=/home/myuser
```

   In this example, the `HOME` environment variable is set to the home directory of the current user. This variable is used by many programs and scripts to **access user-specific data or configuration files.**

5. Localization:

```
LANG=en_US.UTF-8
```

   In this example, the `LANG` environment variable is set to the preferred language and character encoding for the user. This variable is used by many programs and scripts to display information in the appropriate language and format for the user.

   Overall, environment variables can be used in a variety of ways to **store and retrieve information in Unix and Linux systems**. By using environment variables, scripts and programs can be more flexible and customizable, making them more useful and effective for users.

▼ source - Execute commands from a file in the current shell.

The "source" command is a shell built-in command in Linux and Unix-based operating systems. It is used to execute a shell script and load its contents into the current shell context, rather than creating a new sub-shell. Here's an example of how to use the source command:

Let's say you have a shell script called "myscript.sh" that contains the following code:

```
#!/bin/bash

export MY_VAR="Hello, world!"
```

To execute this script using the source command, you would use the following command:

```
source myscript.sh
```

This will load the script "myscript.sh" into the current shell context, executing its contents as if you had typed them directly into the command line.

Now, you can use the environment variable "MY_VAR" in the current shell context. For example, you can print its value using the following command:

```
echo $MY_VAR
```

The output of this command will be:

```
Hello, world!
```

Note that if you had executed the script using the "./myscript.sh" command, it would have created a new sub-shell and exported the environment variable in that sub-shell, rather than the current shell context. In that case, the environment variable would not be available in the current shell context.

Using the source command is useful when you want to execute a shell script and have its contents affect the current shell environment, rather than just creating a new sub-shell with its own environment.

what is the differnce with sh and source command

The `sh` command is used to execute a shell script, while the `source` command is used to execute a script in the current shell environment. The `sh` command runs the script in a new shell, while the `source` command runs the script in the same shell.

when are they both used

The `sh` command is typically used when you want to execute a script in a new shell, while the `source` command is typically used when you want to execute a script in the current shell environment. For example, if you want to execute a script that **changes environment variables,** you would use the `source` command.

▼ alias - Create an alias for a command.

The `alias` command is used to create an alias for a command. This can be used to make a complex command easier to remember or to create a shortcut for a command. For example, you can use the `alias` command to create an alias for the `ls` command as follows:

`alias ll='ls -l'`

This will create an alias called `ll` which is equivalent to running the `ls -l` command. When you run the `ll` command, the output will be the same as if you had run the `ls -l` command.

▼ how to make an alias permanent

To make an alias permanent, you can add it to your shell configuration file. The exact file name and location may vary depending on your operating system and shell.

Here are the general steps:

1. Open your shell configuration file using a text editor. For example, in Bash, the configuration file is usually `~/.bashrc`.
2. Add your alias definition to the file. For example, if you want to create an alias for the `ls` command to always show hidden files, you can add the following line to your `~/.bashrc` file:

   Copy

   ```
   alias ls='ls -a'
   ```

3. Save the file and exit the text editor.
4. Source the shell configuration file to apply the changes to your current shell session. For example, in Bash, you can run the following command:

   Copy

   ~~~
   source ~/.bashrc
   ```

   This will reload the configuration file and make your alias available for use in the current session.
   ~~~

From now on, the alias will be available every time you start a new shell session or open a new terminal window.

▼ unalias - Remove an alias for a command.

The `unalias` command is used to remove an alias that was previously created with the `alias` command. For example, if you had previously created an alias called `ll` with the command `alias ll='ls -l'`, you could remove it with the command `unalias ll`. Running the `unalias` command will not produce any output.

▼ rsync - Remote file synchronization.

`rsync` is a powerful command-line tool used for copying and synchronizing files and directories between two locations on the **same or different computers**. `rsync` is commonly used for **backup and synchronization purposes** because it can copy only the differences between two sets of files, which saves time and network bandwidth.

Here's an overview of some `rsync` variations and examples:

1. Copying files and directories locally:

```
rsync -av /path/to/source /path/to/destination
```

This command copies the contents of the `source` directory to the `destination` directory. The `-a` option preserves the permissions, ownership, timestamps, and links of the files, while the `-v` option enables verbose output.

2. Copying files and directories over SSH:

```
rsync -avz -e ssh /path/to/source user@remote:/path/to/destination
```

This command copies the contents of the `source` directory to the `destination` directory on a remote computer over SSH using the `ssh` protocol. The `-a` option preserves the permissions, ownership, timestamps, and links of the files, while the `-v` option enables verbose output, and the `-z` option compresses the data during transfer.

3. Synchronizing files and directories:

```
rsync -av --delete /path/to/source/ /path/to/destination/
```

This command synchronizes the contents of the `source` directory with the `destination` directory, copying only the differences between the two sets of files. The `--delete` option deletes any files in the destination that do not exist in the source. The trailing slashes after the directory paths are important to ensure that `rsync` copies the contents of the directories rather than the directories themselves.

4. Copying files and directories with exclusions:

```
rsync -av --exclude '*.log' /path/to/source/ /path/to/destination/
```

This command copies the contents of the `source` directory to the `destination` directory, excluding any files with the `.log` extension. The `--exclude` option specifies a pattern to exclude from the transfer.

5. Copying files and directories with bandwidth limitation:

```
rsync --bwlimit=1000 -av /path/to/source/ /path/to/destination/
```

This command copies the contents of the `source` directory to the `destination` directory, limiting the bandwidth used to 1000 kilobits per second. The `--bwlimit` option specifies the maximum transfer rate in kilobits per second.

Possible output for the above examples can include progress information on the transfer, such as the number of files transferred, the transfer rate, and the estimated time remaining. For example:

```
sending incremental file list
./
file1.txt
file2.txt

sent 1234 bytes  received 5678 bytes  789.00 bytes/sec
total size is 1024  speedup is 0.05
```

Overall, `rsync` is a powerful and flexible tool for copying and synchronizing files and directories between two locations. It offers a variety of options and variations to suit different use cases, making it a popular choice for backup and synchronization tasks.

▼ wait - Wait for a process to complete.

In Unix and Linux systems, the `wait` command is used to wait for one or more background processes to complete before continuing with the execution of a shell script or command. The `wait` command is typically used in **shell scripts** to synchronize the execution of multiple processes and ensure that they complete in the correct order.

Here's an example of how to use the `wait` command:

Let's say you have a shell script named `myscript.sh` that launches two background processes, `process1` and `process2`, and you want to wait for both processes to complete before continuing with the script. The contents of the script might look something like this:

```
#!/bin/bash

# Launch process1 in the background
process1 &
pid1=$!

# Launch process2 in the background
process2 &
pid2=$!

# Wait for both processes to complete
wait $pid1 $pid2

# Continue with script execution
echo "Both processes have completed."
```

In this example, the `process1` and `process2` commands are launched in the background using the `&` operator, which allows them to run concurrently with the rest of the script. The `pid1` and `pid2` variables are used to store the process IDs (PIDs) of the two background processes.

After launching the processes, the `wait` command is used to wait for both processes to complete before continuing with the script execution. The `$pid1` and `$pid2` variables are passed as arguments to the `wait` command to specify which processes to wait for.

Once both processes have completed, the script continues with the execution of the `echo` command, which prints a message indicating that both processes have completed.

Possible output for this example could be:

```
Both processes have completed.
```

In this example, the `wait` command is used to synchronize the execution of the two background processes and ensure that they complete in the correct order before continuing with the script execution. By using the `wait` command, you can avoid race conditions and ensure that your scripts run reliably and predictably

▼ sleep - pause a script for a specified amount of time

The `sleep` command is used to **pause a script for a specified amount of time**. For example, you can use the command `sleep 5` to pause the script for 5 seconds. When you run the `sleep` command, it will not produce any output.

It is typically used when you want to **add a delay between commands in a script,** or when you want to wait for a certain amount of time before running the next command in a script.

here are some real-world use cases for the `sleep` command:

- Pausing a script for a few seconds before running a command to give a **web service time to respond**
- Pausing a script for a few seconds before running a command to give a **database time to process a query**
- Pausing a script for a few seconds before running a command to give a **system resource time to be available**
- Pausing a script for a few seconds before running a command to give a **remote server time to respond**

▼ Miscellaneous

▼ Special variables**

Here are some examples of Unix/Linux special variables and their output:

1. $0 - This variable stores the name of the shell or script that is currently running. For example:

```
#!/bin/bash

echo "The name of this script is: $0"
```

Output: If the script is saved as "**test.sh**" and executed from the command line with "`./test.sh`", the output will be "`The name of this script is: ./test.sh".`

2. $1-$9 - These variables store the first nine **command-line arguments** passed to the script or command. For example:

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

```
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
```

Output: If the script is saved as "test.sh" and executed from the command line with "`./test.sh arg1 arg2 arg3`", the output will be "`The first argument is: arg1", "The second argument is: arg2", and "The third argument is: arg3`".

3. $#: This variable stores the number of command-line arguments passed to the script or command. For example:

```
#!/bin/bash

echo "The number of arguments is: $#"
```

Output: If the script is saved as "test.sh" and executed from the command line with "**./test.sh arg1 arg2 arg3**", the output will be "`The number of arguments is: 3`".

4. $*: This variable stores all the command-line arguments passed to the script or command as a single string. For example:

```
#!/bin/bash

echo "The arguments are: $*"
```

Output: If the script is saved as "test.sh" and executed from the command line with ".`/test.sh arg1 arg2 arg3`", the output will be "`The arguments are: arg1 arg2 arg3`".

5. $@: This variable stores all the command-line arguments passed to the script or command as separate strings. For example:

```
#!/bin/bash

echo "The arguments are: $@"
```

Output: If the script is saved as "test.sh" and executed from the command line with "`./test.sh arg1 arg2 arg3`", the output will be "`The arguments are: arg1 arg2 arg3`".

6. **$?: This variable stores the exit status of the last command that was executed. For example:**

```
#!/bin/bash

ls /nonexistent_directory
echo "The exit status is: $?"
```

Output: If the script is saved as "test.sh" and executed, the output will be an error message for the "ls" command, followed by "`The exit status is: 2`".

7. **$$: This variable stores the process ID (PID) of the current shell or script. For example:**

```
#!/bin/bash

echo "The process ID is: $$"
```

Output: If the script is saved as "test.sh" and executed, the output will be "`The process ID is: [PID]`".

8. **$USER: This variable stores the username of the current user. For example:**

```
#!/bin/bash

echo "The current user is: $USER"
```

Output: If the script is saved as "test.sh" and executed, the output will be "`The current user is: [username`]".

9. $HOME: This variable stores the home directory of the current user. For example:

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

```
echo "The home directory is: $HOME"
```

Output: If the script is saved as "test.sh" and executed, the output will be "`The home directory is: /home/[username]`".

10. $PATH: This variable stores the search path for executable files. For example:

```
#!/bin/bash

echo "The search path is: $PATH"
```

Output: If the script is saved as "test.sh" and executed, the output will be the current search path for executable files.

11. $PWD: This variable stores the current working directory. For example:

```
#!/bin/bash

echo "The current working directory is: $PWD"
```

Output: If the script is saved as "test.sh" and executed, the output will be "The current working directory is: /path/to/current/directory".

▼ file - Determine file type.**

The `file` command is used to determine the type of a file. For example, if you have a file called `myfile.txt`, you can use the command `file myfile.txt` to determine the type of the file.

The output of this command would be something like this:

`myfile.txt: [ASCII text](poe://www.poe.com/_api/key_phrase?phrase=ASCII%20text&prompt=Tell%20me%20more%20about%20ASCII%20text.)`

```
[ec2-user@ip-172-31-35-68 ~]$ touch demo.txt
[ec2-user@ip-172-31-35-68 ~]$ echo "dasndkasjhdjkas" >> demo.txt
[ec2-user@ip-172-31-35-68 ~]$ file demo.txt
demo.txt: ASCII text
```

▼ find - Search for files and directories.

What is the find command in Linux?

The `find` command lets you efficiently search for files, folders, and character and block devices.

Below is the basic syntax of the `find` command:

```
find /path/ -type f -name <file-to-search>
```

Where,

- `/path` is the path where file is expected to be found. This is the starting point to search files. The path can also be `/` or `.` which represent root and current directory, respectively.
- `type` **represents the file descriptors. They can be any of the below:**
  - `f` – **Regular file** such as text files, images and hidden files.

    `d` – **Directory**. These are the folders under consideration.

    `l` – **Symbolic link**. Symbolic links point to files and are similar to shortcuts.

    `c` – **Character devices**. Files that are used to access character devices are called character device files. Drivers communicate with character devices by sending and receiving single characters (bytes, octets). Examples include keyboards, sound cards and mouse.

    `b` – **Block devices**. Files that are used to access block devices are called block device files. Drivers communicate with block devices by sending and receiving entire blocks of data. Examples include USB, CD-ROM
- `name` is the name of the file type that you want to search.

Examples of the find command

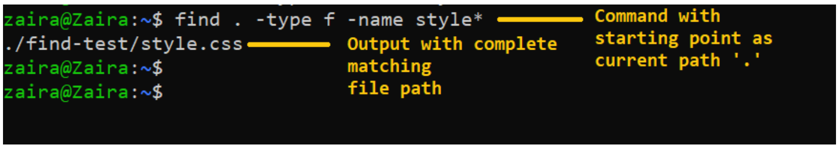
Now we know the syntax of the `find` command, let's look at some examples.

How to search files by name or extension

Suppose we need to find files that contain "style" in their name. We'll use this command:

```
find . -type f -name style*
```

Output



```
zaira@Zaira:~$ find . -type f -name style*
./find-test/style.css
zaira@Zaira:~$
zaira@Zaira:~$
```

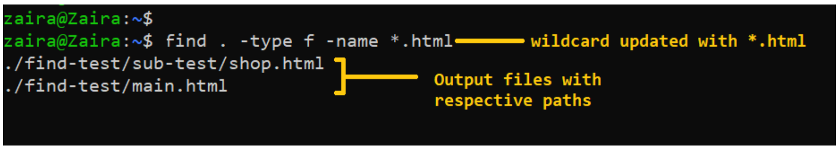
Command with starting point as current path '.'

Output with complete matching file path

Now let's say we want to find files with a particular extension like `.html`. We'll modify the command like this:

```
find . -type f -name *.html
```

Output



```
zaira@Zaira:~$
zaira@Zaira:~$ find . -type f -name *.html
./find-test/sub-test/shop.html
./find-test/main.html
```

wildcard updated with *.html

Output files with respective paths

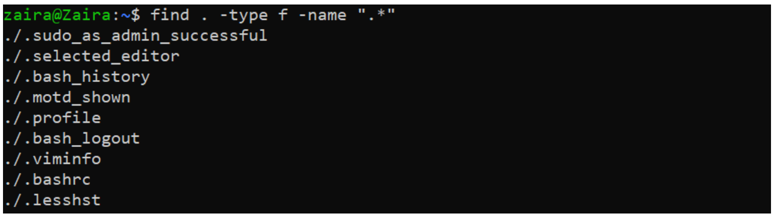
How to search hidden files

Hidden files are represented by a dot in the beginning of the filename. They are normally hidden, but can be viewed with `ls -a` in the current directory.

We can modify the `find` command as shown below to search for hidden files.

```
find . -type f -name ".*"
```

Output



```
zaira@Zaira:~$ find . -type f -name ".*"
./.sudo_as_admin_successful
./.selected_editor
./.bash_history
./.motd_shown
./.profile
./.bash_logout
./.viminfo
./.bashrc
./.lesshst
```

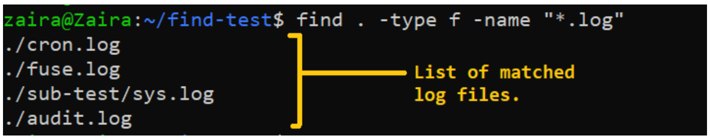
List of hidden files in my home directory

How to search log files and configuration files

Log files usually have the extension `.log`, and we can find them like this:

```
find . -type f -name "*.log"
```

Output



```
zaira@Zaira:~/find-test$ find . -type f -name "*.log"
./cron.log
./fuse.log
./sub-test/sys.log
./audit.log
```

Similarly, we can search for configuration files like this:

```
find . -type f -name "*.conf"
```

How to search other files by type

We can search for character block files by providing `c` to `-type`:

```
find / -type c
```

Similarly, device block files can be found by using `b`:

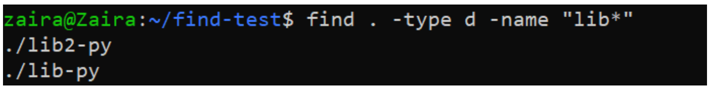
```
find / -type b
```

How to search directories

In the example below, we are finding the folders named `lib`. Note that we are using `-type d`.

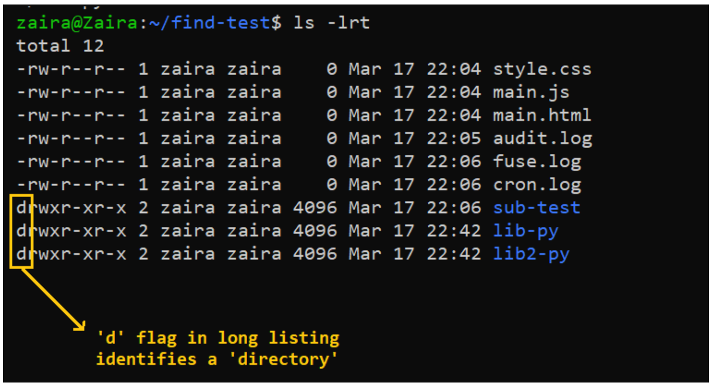
```
find . -type d -name "lib*"
```

Output



```
zaira@Zaira:~/find-test$ find . -type d -name "lib*"
./lib2-py
./lib-py
```

💡 Tip: we can identify directories by looking at the `d` flag in the output of `ls -lrt`.



```
zaira@Zaira:~/find-test$ ls -lrt
total 12
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:04 style.css
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:04 main.js
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:04 main.html
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:05 audit.log
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:06 fuse.log
-rw-r--r-- 1 zaira zaira    0 Mar 17 22:06 cron.log
drwxr-xr-x 2 zaira zaira 4096 Mar 17 22:06 sub-test
drwxr-xr-x 2 zaira zaira 4096 Mar 17 22:42 lib-py
drwxr-xr-x 2 zaira zaira 4096 Mar 17 22:42 lib2-py
```

How to search files by size

An incredibly helpful use of the `find` command is to list files based on a particular size.

`find / -size +250M` Here, we are listing files whose size exceeds 250MB

Other units include:

- `G`: GigaBytes.
- `M`: MegaBytes.
- `K`: KiloBytes
- `b` : bytes.

Just replace <Unit Type> with the relevant unit.

```
find <directory> -type f -size +N<Unit Type>
```

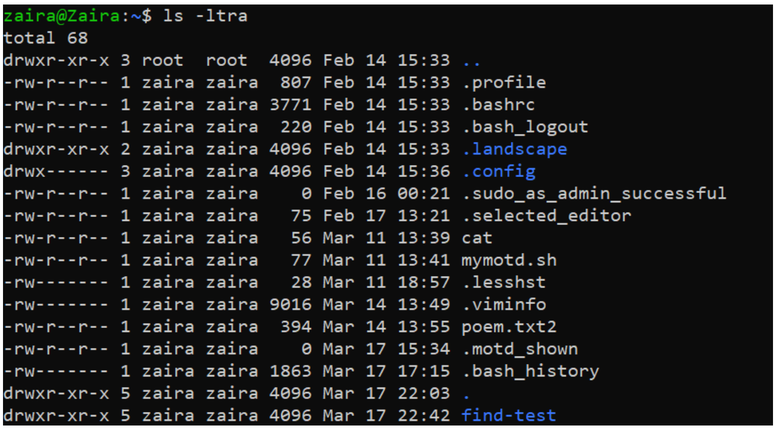
How to search files by modification time

- **mtime +10** means you are looking for a file modified 10 days ago.
- **mtime -10** means less than 10 days.

  ```
  find /path -name -mtime -10 "*.txt"
  ```

- **mtime 10** If you skip + or – it means exactly 10 days.

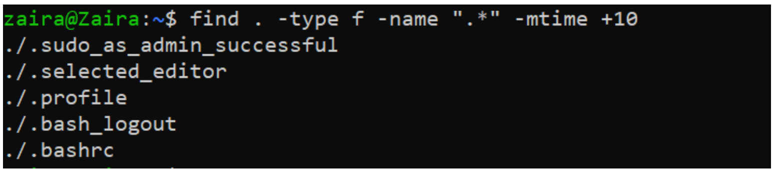
Below are the contents of my home directory:



```
zaira@Zaira:~$ ls -ltra
total 68
drwxr-xr-x 3 root  root  4096 Feb 14 15:33 ..
-rw-r--r-- 1 zaira zaira  807 Feb 14 15:33 .profile
-rw-r--r-- 1 zaira zaira 3771 Feb 14 15:33 .bashrc
-rw-r--r-- 1 zaira zaira  220 Feb 14 15:33 .bash_logout
drwxr-xr-x 2 zaira zaira 4096 Feb 14 15:33 .landscape
drwx------ 3 zaira zaira 4096 Feb 14 15:36 .config
-rw-r--r-- 1 zaira zaira    0 Feb 16 00:21 .sudo_as_admin_successful
-rw-r--r-- 1 zaira zaira   75 Feb 17 13:21 .selected_editor
-rw-r--r-- 1 zaira zaira   56 Mar 11 13:39 cat
-rw-r--r-- 1 zaira zaira   77 Mar 11 13:41 mymotd.sh
-rw------- 1 zaira zaira   28 Mar 11 18:57 .lesshst
-rw------- 1 zaira zaira 9016 Mar 14 13:49 .viminfo
-rw-r--r-- 1 zaira zaira  394 Mar 14 13:55 poem.txt2
-rw-r--r-- 1 zaira zaira    0 Mar 17 15:34 .motd_shown
-rw------- 1 zaira zaira 1863 Mar 17 17:15 .bash_history
drwxr-xr-x 5 zaira zaira 4096 Mar 17 22:03 .
drwxr-xr-x 5 zaira zaira 4096 Mar 17 22:42 find-test
```

Let's apply an example in my home directory.

```
find . -type f -name -mtime +10 ".*"
```



```
zaira@Zaira:~$ find . -type f -name ".*" -mtime +10
./.sudo_as_admin_successful
./.selected_editor
./.profile
./.bash_logout
./.bashrc
```

Here we have files that were modified more than 10 days ago.

find files making it case insenstive

(-iname) used to make the file namecase insenstive

```
find . -type f -iname "Column.txt"
```

```
find . -type f -empty

  finds all the files that are empty

find . -type f -not -empty
find . -type f -! -empty

  Both above commands are used to find NOT empty files
```

```
find . -type f -perm 0777

  find files that has permissions of 777 (here 0 is added by default)

find . -type f -perm 0644
find . -type d -perm 0775
```

-maxdepth and -mindepth

They are used to search files that have exactly same file name (regardless of content inside the files)

For understading purpose let us create file named "note1.txt" at different depths

```
/home/ec2-user/ note1.txt
/home/ec2-user/Directory1 note1.txt
/home/ec2-user/Directory1/Sub_Directory1 note1.txt
/home/ec2-user/Directory1/Sub_Directory1/Sub_sub_Directory1 note1.txt

  here, we have created file named note1.txt in ec2-user, Directory1, Sub_Directory1, Sub_sub_Directory1 (i.e one inside ano
ther and so on)
  This is to create depth dimensions

  Visually:
  Depth       | path
  1.    /home/ec2-user/
  2.    /home/ec2-user/Directory1
  3.    /home/ec2-user/Directory1/Sub_Directory1
  4.    /home/ec2-user/Directory1/Sub_Directory1/

  We have created note1.txt in each Directory one inside another
```

MAX DEPTH

`find . -maxdepth 1 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the current respository i.e /home/ec2-user/

The output will be >>> note1.txt
```

`find . -maxdepth 2 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the current respository i.e /home/ec2-user/ and next depth dimension
 i.e /home/ec2-user/Directory1

The output will be >>>
      note1.txt
      Directory1/note1.txt
```

`find . -maxdepth 3 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the current respository i.e /home/ec2-user/ and next depth dimension
 i.e /home/ec2-user/Directory1 and again /home/ec2-user/Directory1/Sub_Directory1

The output will be >>>
      note1.txt
      Directory1 note1.txt
      Directory1/Sub_Directory1/note1.txt
```

`find . -maxdepth 4 -name "note1.txt"`

```
This follows as explained above.
```

MIN DEPTH

```
The difference from max depth to min depth is that the command searches from backwards i.e from backward dimension. ( 4 > 3
 > 2 > 1)
note: The depth dimension and its levels are fixed i.e

  Depth       | path
  1.    /home/ec2-user/
  2.    /home/ec2-user/Directory1
  3.    /home/ec2-user/Directory1/Sub_Directory1
  4.    /home/ec2-user/Directory1/Sub_Directory1/Sub_sub_Directory1

    -- this will remain same.
```

`find . -mindepth 4 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the 4th dimension i.e /home/ec2-user/Directory1/Sub_Directory1/Sub_su
b_Directory1

The output will be >>>
      /home/ec2-user/Directory1/Sub_Directory1/Sub_sub_Directory1/note1.txt
```

`find . -mindepth 3 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the 4th dimension i.e /home/ec2-user/Directory1/Sub_Directory1/Sub_su
b_Directory1 and 3rd dimension /home/ec2-user/Directory1/Sub_Directory1

The output will be >>>
      Directory1/Sub_Directory1/Sub_sub_Directory1/note1.txt
      Directory1/Sub_Directory1/note1.txt
```

`find . -mindepth 2 -name "note1.txt"`

```
Now if our present working directory is /home/ec2-user/
The above command searches the file named note1.txt at the 4th dimension path, 3rd dimension path and 2nd dimension path

The output will be >>>
      Directory1/Sub_Directory1/Sub_sub_Directory1 note1.txt
      Directory1/Sub_Directory1/note1.txt
      Directory1/note1.txt
```

`find . -mindepth 1 -name "note1.txt"`

```
This follows as explained above.
```

MAX and MIN together is (maybe) similar to Head and Tail command

`find . -maxdepth 3 -mindepth 2  -name "note1.txt"`

output will be >>

```
    Directory1/Sub_Directory1/note1.txt
    Directory1/note1.txt
```

This works like the interection in venn diagram

▼ hostname - Display or set the system hostname.**

In Unix and Linux systems, the `hostname` command is used to display the name of the current host. The `hostname` command is typically used in shell scripts and commands to retrieve the name of the current host, which can be useful for various system administration tasks.

Here's an example of how to use the `hostname` command:

```
$ hostname
```

This command retrieves the name of the current host and prints it to the terminal. The output of the `hostname` command might look something like this:

```
mycomputer.local
```

In this example, `mycomputer.local` is the name of the current host.

You can also use the `hostname` command to set the name of the current host, like this:

```
$ sudo hostname newhostname
```

This command sets the name of the current host to `newhostname`. Note that you must have root privileges (i.e., use the `sudo` command) to change the hostname.

Overall, the `hostname` command is a useful tool for retrieving and setting the name of the current host in Unix and Linux systems. It can be used in shell scripts and commands for system administration tasks and network configuration.

▼ ln - Create a hard or symbolic link.

In Linux, links are used to **create a reference to a file or directory**.

There are two types of links: hard links and soft links (symbolic links).

Hard link

A hard link is a reference to the **same inode** as the original file or directory. This means that changes made to the original file or directory will also be reflected in the hard link. **Hard links can only be created for files or directories that exist on the same file system**. The life of hardlinked file is **independent of original file** i.e even if original file is deleted, the hardlinked file lives along with latest added content

```
[ec2-user@ip-172-31-35-68 ~]$ ll
total 4
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:32 demo.pdf
-rw-r--r--. 1 ec2-user ec2-user 16 Mar 29 17:33 demo.txt
[ec2-user@ip-172-31-35-68 ~]$ touch hard.txt
[ec2-user@ip-172-31-35-68 ~]$ ln hard.txt linkhard.txt
[ec2-user@ip-172-31-35-68 ~]$ ll
total 4
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:32 demo.pdf
-rw-r--r--. 1 ec2-user ec2-user 16 Mar 29 17:33 demo.txt
-rw-r--r--. 2 ec2-user ec2-user  0 Mar 29 17:58 hard.txt
-rw-r--r--. 2 ec2-user ec2-user  0 Mar 29 17:58 linkhard.txt

[ec2-user@ip-172-31-35-68 ~]$ echo "hello" > hard.txt
[ec2-user@ip-172-31-35-68 ~]$ cat hard.txt
hello
[ec2-user@ip-172-31-35-68 ~]$ cat linkhard.txt
hello
[ec2-user@ip-172-31-35-68 ~]$ echo "john" >> hard.txt
[ec2-user@ip-172-31-35-68 ~]$ cat hard.txt
hello
john
[ec2-user@ip-172-31-35-68 ~]$ cat linkhard.txt
hello
john
[ec2-user@ip-172-31-35-68 ~]$ rm hard.txt
[ec2-user@ip-172-31-35-68 ~]$ ll
total 8
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:32 demo.pdf
-rw-r--r--. 1 ec2-user ec2-user 16 Mar 29 17:33 demo.txt
-rw-r--r--. 1 ec2-user ec2-user 11 Mar 29 17:59 linkhard.txt
[ec2-user@ip-172-31-35-68 ~]$ cat linkhard.txt
hello
john
```

Soft link

A soft link, on the other hand, is a reference to the path of the original file or directory. This means that changes made to the original file or directory will not be reflected in the soft link. Soft links can reference files or directories on the same or different file systems. However, the soft link's life is dependent on original file i.e if original file is deleted, soft linked file is also deleted

```
[ec2-user@ip-172-31-35-68 ~]$ ll
total 4
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:32 demo.pdf
-rw-r--r--. 1 ec2-user ec2-user 16 Mar 29 17:33 demo.txt
lrwxrwxrwx. 1 ec2-user ec2-user  8 Mar 29 17:55 softlink.txt -> test.txt
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:55 test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt
[ec2-user@ip-172-31-35-68 ~]$ echo "1234" >> test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat test.txt
1234
[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt
1234
[ec2-user@ip-172-31-35-68 ~]$ echo "abc" >> test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat test.txt
1234
abc
[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt
1234
abc
[ec2-user@ip-172-31-35-68 ~]$ echo "" > test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat test.txt

[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt

[ec2-user@ip-172-31-35-68 ~]$ echo "final" > test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat test.txt
final
[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt
final
[ec2-user@ip-172-31-35-68 ~]$ rm test.txt
[ec2-user@ip-172-31-35-68 ~]$ ll
total 4
-rw-r--r--. 1 ec2-user ec2-user  0 Mar 29 17:32 demo.pdf
-rw-r--r--. 1 ec2-user ec2-user 16 Mar 29 17:33 demo.txt
lrwxrwxrwx. 1 ec2-user ec2-user  8 Mar 29 17:55 softlink.txt -> test.txt
[ec2-user@ip-172-31-35-68 ~]$ cat softlink.txt
cat: softlink.txt: No such file or directory
```

Here are some common commands used to create and manage hard and soft links in Linux:

1. Creating a hard link:

   ```
   ln [original_file] [hard_link]
   ```

   This command creates a hard link from the original file to the hard link.

2. Creating a soft link:

   ```
   ln -s [original_file] [soft_link]
   ```

   This command creates a soft link from the original file to the soft link.

▼ .bashrc

In Unix and Linux systems, the `.bashrc` file is a shell script that is executed whenever a new interactive Bash shell is started. The `.bashrc` file is located in the user's home directory and contains commands and settings that customize the behavior of the Bash shell.

Here's an overview of the most important things to know about the `.bashrc` file:

1. Purpose: The `.bashrc` file is used to customize the behavior of the Bash shell for a particular user. It can be used to set **environment variables, define aliases, create functions, and perform other system configuration tasks.**
2. Location: The `.bashrc` file is located in the user's home directory. The filename starts with a dot (.) to indicate that it is a hidden file.
3. Syntax: The `.bashrc` file is a shell script written in Bash syntax. It contains a series of commands and settings that are executed whenever a new interactive Bash shell is started.
4. Execution: The `.bashrc` file is executed automatically by the Bash shell whenever a new interactive shell is started. This typically happens when the user logs in to the system, but can also happen when a new terminal window or tab is opened.

5. Customization: The `.bashrc` file can be customized by adding or modifying commands and settings to suit the user's needs. Some common customizations include setting environment variables, defining aliases, creating functions, and configuring the prompt.
6. System-wide settings: In addition to the user-specific `.bashrc` file, there is also a system-wide Bash configuration file located at `/etc/bashrc` or `/etc/bash.bashrc`. This file contains settings and commands that apply to all users on the system.

Overall, the `.bashrc` file is a powerful tool for customizing the behavior of the Bash shell in Unix and Linux systems. By understanding how to use the `.bashrc` file, you can create a more efficient and personalized shell environment that suits your needs.

▼ locate - Find files by name.

The `locate` command is used to search for files and directories on the system. For example, if you want to find all files that contain the word "example" in the `/home/user` directory, you could use the command `locate -r '/home/user/.*example.*'`. This command would search the `/home/user` directory and its subdirectories for files that contain the word "example". The output of this command would be a list of paths to all the files that were found.

▼ locate vs find commad

The `locate` and `find` commands are both used to search for files and directories in a Linux or Unix system, but they have some differences in how they work.

`locate` is a command that searches an index of file and directory names on the system, rather than searching the file system itself. The index is typically updated daily by a background process, which means that `locate` can be faster than `find` for finding files and directories. However, because `locate` uses an index, it may not always show the most up-to-date information about the file system.

Here's an example usage of `locate`:

```
locate myfile.txt
```

In this example, `locate` is used to search for any files or directories with the name `myfile.txt`.

`find`, on the other hand, searches the file system in real-time, starting from a specified directory and recursively searching all subdirectories. `find` provides more advanced search capabilities than `locate`, allowing you to search for files based on various criteria such as name, size, and modification time.

Here's an example usage of `find`:

```
find /home/user -name myfile.txt
```

In this example, `find` is used to search for any files or directories with the name `myfile.txt` in the `/home/user` directory and its subdirectories.

In summary, `locate` is faster but may not always show the most up-to-date information, while `find` is slower but provides more advanced search capabilities and real-time file system information.

▼ man - Display manual pages.

The `man` command is used to view the manual page for a command. For example, if you want to view the manual page for the `ls` command, you can use the command `man ls`. This command will display the manual page for the `ls` command. The output of this command will be the manual page for the `ls` command.

▼ mount - Mount a file system. **

explain mount system like i am ten

Sure, I can explain the concept of mounting a file system in a way that a ten-year-old can understand!

Imagine that you have a big toy box with all of your toys inside. You can easily see and play with all of your toys because they are in the toy box, which is like a container that holds everything together.

Now, imagine that you have another box with more toys inside, but this box is hidden in your closet and you can't see or play with the toys inside. That's because the toys are not in your toy box, which is where you usually keep your toys.

In order to play with the toys in the hidden box, you need to take the box out of the closet and put it on the floor where you can see it and reach inside to grab the toys. This is like "mounting" the box of toys to your toy box, which means that you are making the contents of the hidden box accessible and visible to you.

In a computer system, a file system is like a box of files, documents, and other information that is stored on a storage device, such as a hard drive or USB drive. When you "mount" a file system, you are making the contents of the file system accessible and visible to the computer, so that you can read, write, and modify the files and information stored on the device.

When you mount a file system in a computer, you are telling the operating system where the file system is located and how it should be accessed. This allows you to use the files and information stored on the device as if they were part of your local file system, just like you can play with the toys in the hidden box as if they were part of your toy box.

I hope that helps you understand the concept of mounting a file system in a computer system!

In Unix and Linux systems, the `mount` command is used to mount file systems and devices to the file system hierarchy. The `mount` command is a powerful tool that allows you to access and use different types of file systems and devices, such as hard disks, USB drives, network file systems, and more.

Here's an overview of the most important things to know about the `mount` command:

1. Purpose: The `mount` command is used to mount file systems and devices to the file system hierarchy. This allows you to access and use the contents of the file system or device as if it were part of the local file system.
2. Syntax: The basic syntax of the `mount` command is as follows:

   ```
   mount [options] device_or_filesystem mount_point
   ```

   where `device_or_filesystem` is the device or file system to be mounted, and `mount_point` is the directory where the file system or device will be mounted.
3. Options: The `mount` command supports a wide range of options that control the behavior of the command. Some common options include:
   - `o`: Allows you to specify additional mount options, such as read-only mode or file system type.
   - `t`: Specifies the file system type.
   - `r`: Mounts the file system in read-only mode.
   - `w`: Mounts the file system in read-write mode.
   - `a`: Mounts all file systems listed in `/etc/fstab`.
4. Examples: Here are some examples of how to use the `mount` command:
   - Mount a USB drive:

     ```
     mount /dev/sdb1 /mnt/usb
     ```

     This command mounts the first partition of the USB drive located at `/dev/sdb1` to the directory `/mnt/usb`.
   - Mount a network file system:

     ```
     mount -t nfs server:/share /mnt/nfs
     ```

     This command mounts the NFS file system located on the server at the share directory to the directory `/mnt/nfs`.
   - Mount all file systems listed in `/etc/fstab`:

     ```
     mount -a
     ```

     This command mounts all file systems listed in the `/etc/fstab` file.

Overall, the `mount` command is a powerful tool for mounting file systems and devices in Unix and Linux systems. By understanding how to use the `mount` command and its options, you can access and use a wide range of file systems and devices in your system.

▼ fstab **

In Unix and Linux systems, the `/etc/fstab` file is used to define the file systems and devices that should be mounted at boot time.

```
[ec2-user@ip-172-31-35-68 ~]$ cat /etc/fstab
#
UUID=95a6d892-872c-42a1-b366-b20989fe58fc     /           xfs    defaults,noatime  1   1
UUID=EFF7-5416        /boot/efi       vfat    defaults,noatime,uid=0,gid=0,umask=0077,shortname=winnt,x-systemd.automount 0
```

The `fstab` file is a configuration file that contains information about how file systems and devices should be mounted, including the mount point, file system type, and mount options.

Here's an overview of the most important things to know about the `fstab` file:

1. Purpose: The `fstab` file is used to define the file systems and devices that should be mounted at boot time. This allows you to specify the file systems and devices that should be automatically mounted when the system starts up.
2. Location: The `fstab` file is located at `/etc/fstab`. It is a plain text file that can be edited using a text editor.
3. Syntax: The `fstab` file is a tab-separated file that contains information about each file system or device that should be mounted. The fields in the `fstab` file are:

   ```
   device mount_point file_system_type options dump pass
   ```

   where:

   - `device`: The device or file system to be mounted, specified by its device file or network location.
   - `mount_point`: The directory where the file system or device will be mounted.
   - `file_system_type`: The type of file system or device being mounted, such as `ext4` or `nfs`.
   - `options`: The mount options to be used when mounting the file system or device, such as `rw` or `ro`.
   - `dump`: A flag that indicates whether the file system or device should be backed up using the `dump` command.
   - `pass`: The order in which the file system or device should be checked for errors during boot time.

Overall, the `fstab` file is a powerful tool for defining the file systems and devices that should be mounted at boot time in Unix and Linux systems. By understanding how to use the `fstab` file and its syntax, you can customize the behavior of the system and make sure that the necessary file systems and devices are mounted automatically at startup.

▼ package management related

1. `apt-get`: A command-line tool used to install, update, and remove packages on Debian and Ubuntu-based systems.

   ### ▼ Syntax

   Here is the basic syntax of the `apt-get` command:

   ```
   apt-get [options] command [package(s)]
   ```

   In this syntax, `options` are command-line options that modify the behavior of the `apt-get` command, `command` is the action to be performed (e.g., `install`, `upgrade`, `remove`, etc.), and `package(s)` is the name of the package(s) to be affected by the command.

   Here are some common `apt-get` commands with examples:

   1. Install a package:

      ```
      sudo apt-get install PACKAGE_NAME
      ```

      This command installs the specified package and its dependencies.
   2. Upgrade packages:

      ```
      sudo apt-get upgrade
      ```

This command upgrades all installed packages to their latest versions.

3. Remove a package:

   ```
   sudo apt-get remove PACKAGE_NAME
   ```

   This command removes the specified package and its configuration files.

4. Search for a package:

   ```
   apt-cache search SEARCH_TERM
   ```

   This command searches for a package that matches the specified search term.

5. Clean up package cache:

   ```
   sudo apt-get clean
   ```

   This command deletes the package cache from the local repository.

6. Show package information:

   ```
   apt-cache show PACKAGE_NAME
   ```

   This command displays detailed information about the specified package.

These are some of the most common `apt-get` commands with their basic syntax and examples. By understanding how to use these commands, you can easily install, upgrade, and remove packages on your Debian-based Linux system.

2. `yum`: A command-line tool used to install, update, and remove packages on Red Hatbased systems.
3. `rpm`: A command-line tool used to install, upgrade, and remove individual packages on Red Hat-based systems.
4. `dpkg`: A command-line tool used to install, upgrade, and remove individual packages on Debian-based systems.
5. `snap`: A package manager used for installing and managing software packages in a sandboxed environment on various Linux distributions.

▼ difference between apt-get and dpkg

The `apt-get` and `dpkg` commands are both used for package management in Debian-based Linux distributions, but they differ in their functionality and usage.

Here's a brief overview of the differences between `apt-get` and `dpkg`:

1. Functionality: `apt-get` is a higher-level tool that provides a more user-friendly interface for managing packages, while `dpkg` is a lower-level tool that is used for installing, upgrading, and removing individual packages.
2. Dependency handling: `apt-get` automatically handles dependencies when installing or upgrading packages, while `dpkg` does not. This means that you may need to manually resolve dependencies when using `dpkg`.
3. Repository management: `apt-get` automatically downloads packages from configured repositories, while `dpkg` **only installs packages from local files or directories.**
4. Command syntax: `apt-get` uses a simpler and more concise syntax than `dpkg`, making it easier to use for most users.
5. User interface: `apt-get` provides a more user-friendly interface with progress bars, download speeds, and other information, while `dpkg` provides minimal feedback during package installation or removal.

Overall, `apt-get` is a more convenient and user-friendly tool for managing packages and dependencies in Debian-based Linux distributions, while `dpkg` is a lower-level tool that is useful for more advanced users who need to install or remove individual packages manually.

▼ Perform patch and vulnerability scan?

OpenVAS (Open Vulnerability Assessment System) is a free and open-source vulnerability scanner that can be used to scan networks and hosts for security vulnerabilities. OpenVAS is an efficient and powerful tool that provides a comprehensive security assessment of your network, servers, and web applications.

Here are the steps to use OpenVAS:

1. Install OpenVAS: OpenVAS is available on several Linux distributions including Ubuntu, Debian, CentOS, and Fedora. You can install OpenVAS by following the instructions for your specific distribution. For example, on Ubuntu, you can use the following command:

```
sudo apt-get install openvas
```

1. Configure OpenVAS: After installation, you need to configure OpenVAS before you can use it. Run the following command to initialize the OpenVAS database:

```
sudo openvas-setup
```

This command will take some time to complete and will download and install the required software components.

1. Start OpenVAS: After the setup is complete, start OpenVAS by running the following command:

```
sudo openvas-start
```

This will start the OpenVAS scanner, web interface, and other components.

1. Access the OpenVAS web interface: Open a web browser and navigate to `https://localhost:9392`. You will be prompted to enter the default username and password, which are `admin` and `admin` respectively.
2. Create a target: To perform a scan, you need to create a target. A target is simply the IP address or hostname of the system or network you want to scan. In the OpenVAS web interface, go to `Configuration` > `Targets` and click on `Create Target`.
3. Create a task: After creating a target, you need to create a task. A task is a set of instructions that tells OpenVAS what to scan and how to scan it. In the OpenVAS web interface, go to `Scans` > `Tasks`, and click on `Create Task`.

   .

      .

   <u>To create a task in OpenVAS, follow these steps:</u>

   1. Log in to the OpenVAS web interface using your credentials.
   2. In the left-hand menu, click on "Scans" and then click on "Tasks".
   3. Click on the "Create Task" button.
   4. On the "New Task" page, give the task a name and select the target you want to scan. You can choose from one of the targets you have created previously, or you can create a new target by clicking on the "New Target" button.
   5. Under the "Config" tab, configure the scan options for the task. You can choose from a variety of scan options, including port scanners, vulnerability scanners, and web application scanners. You can also specify the scan intensity and set other advanced options.
   6. Under the "Schedule" tab, specify when you want the task to run. You can choose to run the task immediately, or you can schedule it to run at a later time. You can also set the task to run on a recurring basis, such as daily, weekly, or monthly.
   7. Under the "Reports" tab, specify how you want the scan results to be displayed. You can choose to have the results displayed in the OpenVAS web interface, or you can have them emailed to you or saved to a file.
   8. Once you have configured the task settings, click on the "Save" button to create the task.
   9. To run the task, go back to the "Tasks" page and locate the task you just created. Click on the "Play" button next to the task to start the scan.
   10. Once the scan is complete, you can view the results by clicking on the "Reports" button next to the task. The report will show you any vulnerabilities that were found, along with recommendations for remediation.

   That's it! You have now created and run a task in OpenVAS. OpenVAS is a powerful tool that can help you identify security vulnerabilities in your network and systems, so it's important to use it regularly to stay on top of potential security issues.

4. Run the scan: After creating a task, you can start the scan by clicking on the `Play` button next to the task. The scan may take some time to complete, depending on the size of the target and the complexity of the scan.
5. View the results: After the scan is complete, you can view the results by going to `Scans` > `Reports` and clicking on the report for the scan you just ran. The report will list any vulnerabilities that were found, along with recommendations for remediation.

OpenVAS is a powerful tool that can help you identify security vulnerabilities in your network and systems. By following these steps and staying vigilant about security, you can help protect your systems against known vulnerabilities and threats.

Open source vulnerability assessment tool OpenVAS

OpenVAS is a full-highlighted vulnerability scanner. Its abilities incorporate unauthenticated testing, validated testing, different…

https://medium.com/purple-team/open-source-vulnerability-assessment-tool-openvas-e7036c1e48fd

▼ **What is SSHPass and how to use it?**

SSHPass is a utility that allows you to provide a password to SSH commands in a non-interactive way. This is useful when you need to automate SSH connections or run scripts that require SSH authentication, as it eliminates the need for manual password entry. SSHPass works by passing the password to the SSH command using a command-line argument.

Here's how to use SSHPass:

1. Install SSHPass on your local machine. SSHPass is available for most Unix-based systems, including Linux and macOS. You can install it using your package manager or by downloading the source code and compiling it manually. For example, on Ubuntu or Debian, you can install SSHPass using the following command:

   ```
   sudo apt-get install sshpass
   ```

2. Once SSHPass is installed, you can use it to automate SSH connections. To do this, you need to pass the password to the SSH command using the `p` option, followed by the password itself. For example, to connect to a remote machine using SSH and provide the password in a non-interactive way, you can use the following command:

   ```
   sshpass -p yourpassword ssh username@remote-machine
   ```

   Replace `yourpassword` with the actual password you want to use to authenticate the SSH connection, `username` with the username you want to use to connect to the remote machine, and `remote-machine` with the hostname or IP address of the remote machine.

   When you run this command, SSHPass will pass the password to the SSH command, allowing it to authenticate the connection without requiring manual password entry.

3. If you need to run multiple SSH commands that require authentication, you can use SSHPass in combination with shell scripts or other automation tools to automate the process. For example, you could create a shell script that contains a series of SSH commands and use SSHPass to provide the password for each command.

   Note that while SSHPass can be useful for automating SSH connections, it is not a secure method of authentication. Storing passwords in plain text or passing them via command-line arguments can be a security risk, as they can be easily intercepted or viewed by other users on the system. For secure automation of SSH connections, it's recommended to use public-key authentication or other secure authentication methods.

Introduction