Linux Commands: Permissions, Files & Search

Command Line Usage

1. Explanation of file permissions (read, write, execute) and ownership (user, group).
2. Modifying file permissions and ownership (chmod, chown, chgrp).
3. Viewing file content (cat, less, more, head, tail).
4. Searching for files and directories (find, locate).
5. Basic file editing using a command line editor (nano)

Permissions in Linux :

Introduction :
Imagine a house with rooms (directories) that contain different kinds of boxes (files). Every room has a label indicating who can:

- Enter and look inside (read).
- Change the contents inside or rearrange the boxes (write).
- Execute or use the items (execute).

In this house, three types of people can interact with rooms and boxes:

- **Owner:** The person who owns the house (usually the one who created the file or directory).
- **Group:** A selected group of friends whom the owner trusts.
- **Others:** Everyone else who might visit the house.

Types of Permissions:

- Read (r): Like reading a book. You can view the contents but can't alter it.
- Write (w): Like having a pen while reading. You can change the contents.
- Execute (x): For files, it's like a toy – you can 'play' (run) it. For directories, it allows you to enter and access its content.

Viewing Permissions: The "ls -l" Command :-

If we run the command **ls -l** in a directory, it might show something like:

```
drwxr-xr-x 2 user group 4096 Apr 5 12:34 Documents
-rw-r--r-- 1 user group   25 Apr 5 12:33 notes.txt
```

Breakdown:

- The first character: **d indicates a directory. - indicates a file.**
- Next three characters: Permissions for the owner.
- Middle three characters: Permissions for the group.
- Last three: Permissions for others.

In our example:

- **Documents** is a directory. The owner can read, write, and execute (enter it). The group can read and execute (enter it). Others can also read and execute.
- notes.txt is a file. The owner can read and write. Both the group and others can only read.

Changing Permissions: The "chmod" Command :

To change the permissions of a file or directory, use **chmod**.

Symbolic Mode :

- **Granting Permissions using Symbolic Modes :**

Examples :

1. Give read permission to the owner of the file: `chmod u+r filename.txt`
2. Give read and write permissions to the owner and group of the file: `chmod u+rw,g+rw filename.txt`
3. Give execute permissions to everyone: `chmod a+x script.sh`
4. Remove read and execute permissions from others: `chmod o-rx filename.txt`
5. Set the owner to have read and write permissions, but no execute permission, and give only read permissions to group and others: `chmod u=rw,g=r,o=r filename.txt`

- Numeric (Octal) Mode:

  Each permission can be represented as a number:
  - 4 represents **Read (r)**
  - 2 represents **Write (w)**
  - 1 represents **Execute (x)**

Examples :

- Give read, write, and execute permissions to the owner, and read and execute permissions to the group and others :

```
chmod 755 filename.txt
```

- Give read and write permissions to the owner, and only read permissions to the group and others:

```
chmod 644 filename.txt
```

- Give full permissions to the owner, write and execute permissions to the group, and no permissions to others:

```
chmod 761 filename.txt
```

- Recursively Change Permissions on Directories:
  - Give read, write, and execute permissions to the owner of all files and directories, recursively:

```
chmod -R 700 /path/to/directory
```

- Remove All Permissions from a File:

```
chmod 000 filename.txt
```

CHOWN command :

The **chown** command in Linux allows us to change the owner and/or group of a file or directory. When focusing on changing just the group, here's how it works:

- If we want to change the group ownership of a file named example.txt to the group developers :

```
chown :developers example.txt
```

- If we want to change the group ownership of a directory named **project_dir** to the group developers :

```
chown :developers project_dir
```

- To recursively change the group ownership for a directory and all its contents (subdirectories and files), you can use the -R option.

  For example, to change the group ownership of the directory project_dir and all its contents to the group developers:

```
chown -R :developers project_dir
```

- To change the owner of example.txt to alice and the group to developers:

```
chown alice:developers example.txt
```

Introduction to the cat Command :-

The cat (short for "**concatenate**") command in Linux is one of the most frequently used commands. It's used to:

- Display the content of files
- Concatenate files and redirect output in terminal or files
- Append content of a file to another file
- Create a new file

Examples :

- To view the contents of a file, simply type cat followed by the filename :

```
cat filename.txt
```

- We can also use cat to view the content of multiple files at once :

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

- We can create a new file using cat :

```
cat > newfile.txt
```

After running this command, you can start typing the contents of the file on the terminal. When you're done, press Ctrl + D to save the file and exit.

- We can append the contents of one file to another

```
cat file1.txt >> file2.txt
```

- If we want to display the contents of a file with line numbers, **use the -n option**:

```
cat -n file1.txt
```

Introduction to less Command :-

The **less** command in Linux is used to view the content of a file one screen at a time. It's similar to the **more** command but provides more advanced features. One of the most significant benefits of **less** over **more** is that **less** allows both forward and backward navigation through the file.

```
less filename.txt
```

This opens filename.txt in less for viewing. You can scroll through the file using the arrow keys.

Searching Within less

While viewing a file in less, you can search for a pattern:

Forward search: Press / followed by the pattern and press Enter. For example, /pattern searches for "pattern".

Backward search: Press ? followed by the pattern.

After searching, press **n** to find the next occurrence or **N** for the previous occurrence.

Go to a Specific Line
While inside less:

```
42g
```

This takes you to the 42nd line.

Jump to End or Beginning of a File
While inside less:

G will take you to the end of the file.
g will take you to the beginning.

Exiting less : Press q to quit less.

HEAD and TAIL Command :

head and **tail** commands are useful for quickly viewing parts of files. As their names suggest, **head shows the beginning (or head)** of files, while **tail displays the end (or tail)** of files.

Examples of head command :-

1. Display the first 10 lines of a file named sample.txt:

```
head sample.txt
```

2. Display the first 5 lines of sample.txt:

```
head -n 5 sample.txt
```

The -n option allows you to specify the number of lines you wish to view.

Examples of tail command :-

1. Display the last 10 lines of a file named sample.txt:

```
tail sample.txt
```

2. Display the last 5 lines of sample.txt:

```
tail -n 5 sample.txt
```

GREP COMMAND :

Introduction : **grep** stands for "**Global Regular Expression Print**". It is a powerful tool for searching specific patterns in files.

Usage: Describe the basic use-case. It's commonly used to find specific strings or patterns in files.

Simple String Search:

```
grep "pattern" filename
```

Use an example with a small file and show how grep highlights the matching pattern.

Case-insensitive search: -i option.

```
grep -i "pattern" filename
```

Search across multiple files:

```
grep "pattern" file1 file2 file3
```

Using Flags with grep :

- **Line numbers with matches:**

  ```
  grep -n "pattern" filename
  ```

- **Display only count of matching lines:**

```
grep -c "pattern" filename
```

- **Inverse the match : Give the lines which don't contain the pattern.**

  ```
  grep -v "pattern" filename
  ```

- **Search Recursively: Use -r or -R to search for a word recursively in directories: grep -r "word" directory/**

- **Match whole word :**
  ```
  clear
  grep -w "good" example.txt
  ```

Using Regular Expressions :

- Search for lines that start with a particular word:

  ```
  grep "^word" filename.txt
  ```

- lines that end with a particular word:

  ```
  grep "word$" filename.txt
  ```

- Wildcard Character:

  . [DOT] can be used to represent any single character:

  ```
  grep "w.rd" filename.txt
  ```

  This will match "word", "ward", "w3rd", etc.

WC Command :

wc stands for **"word count."** It allows users to get the number of lines, words, and bytes or characters from a text source, whether it be a file or an output from another command.

```
wc filename.txt
```

The output will be in the format of:

[lines] [words] [bytes] filename.txt

Count Lines:

Use the -l option to count just the lines:

```
wc -l filename.txt
```

Count Words:

Use the -w option to count only words:

```
wc -w filename.txt
```

Count Characters:

The -c option allows you to count characters (bytes):

```
wc -c filename.txt
```

Real-world Usage & Combining with Other Commands:

Piping with echo:

Students can understand how wc works with piped input:

```
echo "This is a test." | wc -w
```

This should return 4 because there are four words.

Piping in Linux:

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow the stdout of a command to be connected to the stdin of another command. You can make it do so by using the **pipe character '|'.**

Piping with cat:

Count words across multiple files:

```
cat file1.txt file2.txt | wc -w
```

Piping with grep:

-o option with grep : Print only the matched parts of a matching line, with each such part on a separate output line.

By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the **-o option**.

grep -o “good” example.txt

Count the total number of occurrences of a word in a file:

```
grep -o "word" input | wc -l
```

Q: Print line numbers from 5 to 10 from a file.

To print lines from x to y (e.g., from 5 to 10):

```
head -10 filename | tail -6
```

Explanation : head -10 filename prints the first 10 lines. From those, tail -6 then prints the last 6 lines, which correspond to lines 5 through 10.

Search apple or banana in a file :

```
grep -E "apple|banana" filename
```

Lines that start with "apple"

```
grep -E "^apple" filename
```

Lines that end with "apple"

```
grep -E "apple$" filename
```

SED and AWK Commands

sed (Stream Editor):

sed is a stream editor that is used to perform basic text transformations on an input stream **(a file or input from a pipeline).**

Basic Usage:

- Replace first occurrence of a string in a line: **sed 's/find/replace/' filename**

```
echo "apple apple apple" | sed 's/apple/orange/'

-> This will output "orange apple apple". It replaced only the first
occurrence. To replace all occurrences on a line, use the g option.
```

- Replace all occurrences of a string in a line: **sed 's/find/replace/g' filename**

```
echo "apple apple apple" | sed 's/apple/orange/g'

-> This will output "orange orange orange".
```

- Replace on a specific line number:

```
sed '3 s/find/replace/' filename
```

Intermediate Usage:

- Delete lines from a file: sed '3,5d' filename deletes lines 3 to 5.
- Print only specific lines: sed -n '3,5p' filename prints lines 3 to 5.

Advanced Usage:

- Perform case **insensitive search:**

```
sed 's/find/replace/gI' filename
```

- Replace in place in a file (altering original file):

```
sed -i 's/find/replace/g' filename
```

- Replace in a specific line:

```
sed '2s/orange/grape/'
```

Now let's discuss how sed works:

Read: The sed command reads a line of input from its source one line at a time into an area of memory known as the pattern space. The input is typically a file, the output of another command, or input from a terminal.

Execute: Once the line is read, sed executes a set of instructions that you have passed to it. These instructions are applied on the line read into the pattern space. The commands can change the line, delete it, print it out, or store it in a hold buffer for further action.

Display: After it has applied all the commands on the pattern space, sed outputs the line to standard output (usually the terminal, unless redirected). The line is then deleted from the pattern space, and the process is repeated from step 1 with the next line.

Here's an example to illustrate this:

Suppose we have a file named file.txt with the following contents:

```
Hello World
Goodbye World
Hello again
```

Let's say we want to **replace all occurrences of 'Hello' with 'Hi'**. Here is how we can do it using sed:

```
sed 's/Hello/Hi/g' file.txt
```

In this example, **'s/Hello/Hi/g'** is the command we are passing to sed. Here's what it does:

's' stands for **substitute**.
'Hello' is the search pattern – the text we want to replace.
'Hi' is the replacement text – what we want to replace 'Hello' with.
'g' stands for global, which means replace all occurrences on a line, not just the first one.

And the output will be:

```
Hi World
Goodbye World
```

```
Hi again
```

Remember, unless we specify **-i** as an option (like sed -i 's/Hello/Hi/g' file.txt), sed doesn't change the original file. It sends the output to the standard output (the terminal, usually). If you want to save changes to a file,
you should redirect the output to a file.

If you want to edit the file in-place **(i.e., change the original file)**, you can use the -i option with sed:

```
sed -i 's/Hello/Hi/g' file.txt
```

This changes the input file directly. Be cautious when using -i, because it overwrites the original file. It's a good idea to keep a backup of your original file when using this option.

Here's how we can save the output of sed command to a file :

```
sed 's/Hello/Hi/g' file.txt > outputfile
```

If outputfile doesn't exist, the **redirection operator >** creates it. If it does exist, **> overwrites it.** To append the output to the file without overwriting existing content, use the **>> operator**.

Deleting Lines using sed command:

We can delete specific lines in a file with sed.

For example, to delete the first line:

```
sed '1d' filename
```

To delete the last line:

```
sed '$d' filename
```

To delete a range of lines, like lines 3 through 5:

```
sed '3,5d' filename
```

Inserting and Appending Text:

You can use sed to insert or append text on a line. To insert "Hello" at the beginning of every line:

```
sed 's/^/Hello /' filename
```

And to append "Goodbye" at the end of every line:

```
sed 's/$/Goodbye/' filename
```

Replacing Strings on Specific Lines Only:

If you want to replace a string only on specific lines, you can do that with sed. To replace "find" with "replace" only on lines 2 through 4:

```
sed '2,4s/find/replace/' filename
```

Print Only Certain Lines:

By default, sed prints out all lines. With the **-n option,** you can make it quiet and only print lines you specify. To print only line number 3:

```
sed -n '3p' filename
```

To print lines 3 through 5:

```
sed -n '3,5p' filename
```

Replacing Strings Except on Certain Lines:

sed can also replace strings everywhere except on certain lines. For example, to replace "Hello" with "Hi" everywhere except lines 3 through 5:

```
sed '3,5!s/Hello/Hi/' filename
```

Replacing a Whole Line:

To replace the whole line when a match is found, **use the c command in sed**. The following command will replace lines containing "match" with "Replacement text":

```
sed '/match/c\Replacement text' filename
```

Append Text at a Specific Line :

Use Case: Append "EOF" at the end of a file.

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

The expression **$a** is used to append text after the last line of the input.

Here's a breakdown of the components:

$: This symbol represents the last line of the input stream or file in sed. It's a special address that matches only the last line.

a: a in sed stands for "**append**". It's used to append text after a specific line.

Multiple Operations in a Single Command:

Use Case: Delete lines containing "error", and replace "old" with "new".

```
sed '/error/d; s/old/new/g' file.txt
```

AWK Command :

The **awk** command is a text-processing language that is particularly adept at handling structured data, such as columns of numbers or text. Its operation can be understood in terms of its basic workflow, which revolves around reading input line by line, splitting the input into records, and processing each record based on specified patterns and actions.

Here's a step-by-step breakdown of how awk works:

Reading Input:

By default, awk reads its input line by line, treating each line as a record. This behavior can be changed by adjusting the record separator.
The input can be provided in various ways: directly from the command line, from a file, or via a pipe from another command.

Splitting into Fields:

- Each record (by default, a line) is automatically split into fields.
- The default field separator is white space (like spaces or tabs). This means awk splits a line wherever it sees spaces or tabs.
- The field separator can be changed using the -F option or by setting the FS (Field Separator) variable.
- Individual fields can be accessed using **$1, $2, $3,** etc., where $1 is the first field, $2 is the second, and so on. $0 represents the entire record.

Pattern Matching:

awk scans each record for patterns you've specified. A pattern dictates which records should be acted upon.
If no pattern is given, awk performs the specified action on every record.
Patterns can be regular expressions, relational expressions, or combinations thereof.

Executing Actions:

- For records that match a given pattern, awk performs the associated action, which is typically enclosed in curly braces **{...}**.
- Actions can be any combination of commands, calculations, and functions. If no action is specified for a pattern, the default action is to print the entire record ($0).
- Actions are executed in the order they appear in the command or script.

Built-in Variables:

- awk has several built-in variables like **NR (Number of Records)**, **NF (Number of Fields in the current record)**, **OFS (Output Field Separator)**, and others. These variables can be used to gather insights about the input or to modify the behavior of awk.

End and Begin Blocks:

- awk scripts can have BEGIN and END blocks.

- The BEGIN block is executed before any input is read, which is useful for initializing variables or printing headers.
- The END block is executed after all input has been read, which is handy for summarizing data or performing final calculations.

Processing the Next Record:

After processing an input line, awk automatically reads the next line and starts the process again, continuing until all input has been read.

Use Cases :

- **Printing Entire Content of a File**

Command:

```
awk '{print}' file.txt
```

Working:

The command is simply telling awk to print every line in file.txt. Since no pattern is specified before the action {print}, awk assumes it should operate on all lines. {print} is an action telling awk to print the line as it is. Since print without any field number defaults to printing $0 (the whole line), the entire content of the file is printed.

- **Printing Specific Fields**

Command:

```
awk '{print $1, $3}' file.txt
```

Working:

This command prints the first and third fields of each line in file.txt. awk reads the file line by line, splits each line into fields based on whitespace (or any other specified delimiter), then {print $1, $3} tells awk to print the first and third fields of each line.

- **Print line numbers from 3 to 6**

Command:

```
awk 'NR==3, NR==6 {print NR,$0}' file.txt
```

- **Conditional Printing**

Command:

```
awk '$1 == "example" {print $2}' file.txt
```

Working:

Here, $1 == "example" is a condition that awk checks for each line in file.txt. If the first field of a line is "example", awk performs the {print $2} action, which prints the second field of the line.

- **Line Count Using awk**

Command:

```
awk 'END {print NR}' file.txt
```

Working:

This command prints the total number of lines in file.txt. It does so by using the built-in awk variable **NR**, which keeps a count of the **number of records** (lines, by default) processed. As awk reads each line, it increments NR. At the END, it simply prints the value of NR, which is the total line count.

- **Print the lines with more than 10 characters:**

```
awk 'length($0) > 10' file.txt
```

Note : It counts white spaces as well.

- **Print the content with line numbers**

Use Case: Print the content of the line along with line numbers

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

- **Print Specific Fields**

Use Case: Suppose you have a file named employees.txt containing the name, age, and designation of employees:

```
Alice 30 Developer
Bob 25 Designer
Charlie 35 Manager
```

We want to print only the names of the employees.

Command:

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

Working:

- awk reads the file employees.txt line by line.
- For each line, it splits the data into fields using the default separator (whitespace).
- The action { print $1 } instructs awk to print the first field, i.e., the name of the employee.

Output:

```
Alice
Bob
Charlie
```

- **Sum a Column of Numbers**

Use Case: Given a file sales.txt that lists items and their respective sales:

```
Laptop 1500
Phone 800
Tablet 300
```

We want to find the total sales.

Command:

```
awk 'BEGIN { sum=0 } { sum += $2 } END { print sum }' sales.txt
```

Working:

- The BEGIN block initializes the variable sum to 0 before any line is processed.
- For each line, the value of the second field (the sales figure) is added to the sum.
- After processing all lines, the END block prints the final value of sum.

Output:

```
2600
```

- **Filter Data Based on a Condition**

Use Case: From the employees.txt file, you want to list only those employees older than 28.

Command:

```
awk '$2 > 28' employees.txt
```

Working:

- Each line is read and split into fields.
- The pattern $2 > 28 checks if the second field (age) is greater than 28.
- If the condition is true, the default action of printing the entire line ($0) is executed. If no action is specified after a pattern, the default action is to print the line.

Output:

```
Alice 30 Developer
Charlie 35 Manager
```

- **Modify Field Data**

Use Case: For the sales.txt file, you want to increase the sales of each item by 10%.

Command:

```
awk '{ $2 = $2 * 1.10; print $0 }' sales.txt
```

Working:

- Each line's second field is updated to be 10% greater.
- The entire updated line is then printed using print $0.

Output:

```
Laptop 1650
Phone 880
Tablet 330
```

How to give different delimiter than space :

```
awk -F':' ..............
```

- **Calculate Average**

Use Case: Calculate the average of numbers in the second column.

```
awk '{ total+=$2; count++ } END { print "Average:", total/count }' data.txt
```

- Data Extraction and Transformation

Use Case: Convert a space-separated values file to CSV format.

```
awk '{OFS=","; print $1, $2, $3}' data.ssv > data.csv
```

- Write a linux command to display lines that have the same word at the beginning and the end, in a file.

```
awk '{ if ($1 == $NF) print }' input
```

Content :

- What are log files?
- Why is Log analysis important?
- Log Parsing and Extraction.
- Log Filtering, Searching & analysis - grep, awk, sed, sort & uniq commands.
- Common Log Analysis Scenarios :
    - Error Analysis.
    - Performance Analysis.
    - Security Analysis.

Recap :

Quiz : Write a command to count the number of occurrences of word error in a log file.

```
grep -c "error" logfile.txt
```

Quiz : Command to check the real time log entries.

```
tail -f logfile.txt
```

Quiz : Command to analyze Disk Space Usage.

```
du <directory name>
```

du command stands for **disk usage**.
With **du** command we can use **-h** option to give the output in human readable format.

SORT Command :-

Sorting is a fundamental operation in data manipulation and analysis. By default, the entire input is taken as the sort key. Blank space is the default field separator.

Note : Sort command does not actually change the input file.

1. Sort the lines of the file alphabetically.

```
sort file.txt
```

2. Sort lines in reverse order (from highest to lowest).

```
sort -r file.txt
```

3. Numerical Sorting

```
sort -n file.txt
```

4. Case-Insensitive Sorting

```
sort -f file.txt
```

5. **-nr option:** To sort a file with numeric data in reverse order we can use the

```
sort -nr file.txt
```

6. **-k Option**: Sort a file on the basis of any column number by using -k option. For example, use “-k 2” to sort on the second column.

Let’s say our file has 2 columns, name of the student and marks.
Now if we want to sort this file based on the marks.

```
sort -k 2n employee.txt
```

7. **-u option:** To sort and remove duplicates pass the -u option to sort. This will write a sorted list to standard output and remove duplicates.

```
sort -u file.txt
```

Question : Write a command to sort the input file based on the second column (separated by tab).

```
sort -t$'\t' -k2 file.txt
```

Explanation:

- **sort:** sort program.
- **-t$'\t':** This flag specifies the field separator to be used for sorting. The $'\t' is a Bash syntax that represents a tab character, so **-t$'\t'** sets the field separator to a tab.
- **-k2:** This flag specifies the sorting key. Here's what it means: -k2: This specifies the second field as the sorting key.
- **file.txt:** This is the input file containing the data to be sorted.

The sort command is being used to sort the lines in the file.txt file based on the second field, using a **tab (\t)** as the field separator.

Question : Given a file of IP addresses, sort it based on IP addresses in ascending order.

```
sort -t. -k1,1n -k2,2n -k3,3n -k4,4n ip_addresses.txt
```

Explanation:

- **-t.** : This flag specifies the field separator to be used for sorting. In this case, a **period (.)** is used as the separator. It indicates that the IP address should be split into fields based on periods.
- **-k1,1n -k2,2n -k3,3n -k4,4n**: These flags specify the sorting keys. Here's what each part means:
    - **-k1,1n:** This specifies the first field (the first octet) as the sorting key, using numerical sorting (-n).
    - **-k2,2n:** This specifies the second field (the second octet) as the sorting key, using numerical sorting.
    - **-k3,3n:** This specifies the third field (the third octet) as the sorting key, using numerical sorting.
    - **-k4,4n:** This specifies the fourth field (the fourth octet) as the sorting key, using numerical sorting.

The n after each field specification indicates that **numerical sorting** should be used for that field.

Summary :

The sort command is being used to sort IP addresses stored in the ip_addresses.txt file. The -t. flag specifies that the fields are separated by periods, and the -k flags indicate that the sorting should be performed based on the individual octets of the IP addresses. Each octet is treated as a separate sorting key, and numerical sorting is used for each octet.

UNIQ Command :

Linux command used to report or filter out the repeated lines in a file.

uniq command is used to detect the adjacent duplicate lines and also deletes the duplicate lines. **uniq** filters out the adjacent matching lines from the input file and writes the filtered data to the output file.

Syntax :

```
uniq [option] [input_file]
```

Note: uniq command can't detect the duplicate lines unless they are adjacent to each other. The content in the file must be therefore sorted before using uniq or you can simply use sort -u instead of uniq command.

Options with UNIQ command :-

- -c option : It gives the number of times a line was repeated.
- -d option : It only prints the repeated lines, only one per group..
- -D option : It prints only duplicate lines but not one per group.
- -u option : It prints only the unique lines.
- -i option : It is used to make the comparison case-insensitive.

Examples :-

```
uniq sorted_file.txt
```

This command processes a sorted file and removes any consecutive duplicate lines, keeping only the first occurrence of each unique line.

```
uniq -c sorted_file.txt
```

Adding the -c flag will display a count of occurrences next to each unique line in the sorted file.

```
uniq -i sorted_file.txt
```

The -i flag tells uniq to treat uppercase and lowercase characters as equivalent when determining duplicates.

```
uniq -d sorted_file.txt
```

The -d flag will show only the lines that are duplicates in the sorted file.

```
uniq -u sorted_file.txt
```

The -u flag will show only the lines that are unique in the sorted file, effectively filtering out duplicates.

```
sort unsorted_file.txt | uniq
```

uniq works best with sorted data. If your data is unsorted, you should sort it first. To use uniq, it's recommended to sort the data first using the sort command, and then pipe the sorted data into uniq.

```
sort logfile.txt | uniq > deduplicated_log.txt
```

Removing Consecutive Duplicates from a Log file. This command sorts the log file and removes consecutive duplicate lines, creating a new log file without duplicates.

LOG Analysis :

1. Counting Occurrences of Log Messages: To identify and count how many times each log message appears in a log file:

```
sort logfile.txt | uniq -c
```

2. Identifying Most Frequent Log Messages: To find the most common log messages along with their counts:

```
sort logfile.txt | uniq -c | sort -nr
```

3. Finding Unique Log Entries: To identify unique log entries (removing duplicates):

```
sort logfile.txt | uniq
```

Example :

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

```
# Set the log file path
LOG_FILE="/path/to/your/log/file.log"

# Function to analyze logs using grep, sort, and uniq
analyze_logs() {
    echo "Analyzing logs using grep, sort, and uniq..."

    # Search for specific patterns using grep and store the results in a
variable
    error_logs=$(grep "ERROR" "$LOG_FILE")
    warning_logs=$(grep "WARNING" "$LOG_FILE")

    # Count and display the occurrences of each error type
    echo "Error log count: $(echo "$error_logs" | wc -l)"
    echo "Warning log count: $(echo "$warning_logs" | wc -l)"

    # Sort and unique the error messages
    sorted_unique_errors=$(echo "$error_logs" | sort | uniq)
    echo "Sorted and unique error messages:"
    echo "$sorted_unique_errors"

    # Sort and unique the warning messages
    sorted_unique_warnings=$(echo "$warning_logs" | sort | uniq)
    echo "Sorted and unique warning messages:"
    echo "$sorted_unique_warnings"
}

# Main function
main() {
    echo "Log Analysis Script started"

    if [ -f "$LOG_FILE" ]; then
        analyze_logs
    else
        echo "Log file not found: $LOG_FILE"
    fi

    echo "Log Analysis Script completed"
}

# Run the main function
main
```

- This script counts the occurrences of each error type using the wc -l command.
- The error and warning logs are sorted and unique using the sort and uniq commands.
- The script then displays the sorted and unique error and warning messages.

SSH (Secure Shell)

SSH is a network communication protocol that enables two computers to communicate and share data.

SSH is used for remote login to another computer. You can run commands on the remote machine as if you were there, and all communication is encrypted.

Why it is called a Secure Shell : Because the communication between host and client will be encrypted.

- The default port for SSH communication is 22.

Syntax
The basic syntax for SSH is:

```
ssh [username]@[hostname]
```

username: The username on the remote machine you want to log in to.
hostname: The IP address or domain name of the remote machine.

Example
To SSH into a machine with IP address 192.168.1.100 as user john:

```
ssh john@192.168.1.100
```

When prompted, enter the password for john on the remote machine.

SCP (Secure Copy Protocol)

SCP is used for securely transferring files between local and remote machines.

Syntax : For transferring files from a local machine to a remote machine:

```
scp [local-file-path] [username]@[hostname]:[remote-file-path]
```

For transferring files from a remote machine to a local machine:

```
scp [username]@[hostname]:[remote-file-path] [local-file-path]
```

Example : Copy File to Remote Server: To copy a file named file.txt from the local machine to the home directory of john on a remote machine with IP 192.168.1.100:

```
scp file.txt john@192.168.1.100:~
```

Copy File from Remote Server: To copy a file named file.txt from the remote machine's home directory to the local machine:

```
scp john@192.168.1.100:~/file.txt .
```

Practical Tips

Copy Directories: Use the -r flag to copy directories.

```
scp -r local-directory john@192.168.1.100:~
```

PS Command : Process Status

The ps command is used to display information about the running processes in our system.

Example 1: Basic ps command

```
ps
```

When we run ps with no options, it shows the processes that are running in your current terminal session for your user.

```
  PID  TTY          TIME CMD
 2321 pts/1    00:00:00 bash
 2361 pts/1    00:00:00 ps
```

Here, each row represents a process:

- PID is the Process ID
- TTY is the terminal type
- TIME is the total accumulated CPU time since the process has started
- CMD is the command name

This will show a list of processes running in the current terminal for the current user.

Example 2: Show all processes for all users

```
ps aux
```

The **a** flag shows processes for all users, **u** provides detailed information, and **x** includes processes not attached to a terminal.

Displays detailed information about all running processes.

Here's a breakdown of what each column means:

- USER: The owner of the process.
- PID: The Process ID.
- %CPU: CPU utilization of the process.
- %MEM: Memory utilization of the process.
- VSZ: Virtual memory usage.
- RSS: Resident set size.
- TTY: Terminal type.

- STAT: Process status.
- START: Start time of the process.
- TIME: Total CPU time used since the process started.
- COMMAND: Command which started this process.

Example 3 : ps -e: Displays all the running processes.

```
ps -e
```

Example 4 : ps -ef: Shows full format listing.

```
ps -ef
```

Example 5 : ps -u [username]: Shows all processes for a particular user.

```
ps -u username
```

Problem Statement : Find Processes Related to a Particular Application

If we want to find out the status of a certain application, let's say Java, then :

```
ps aux | grep java
```

Once we find the PID, we can use ps to find more details about the process:

```
ps -p [PID]
```

Problem Statement : Listing Your Own Processes
To list all your processes, you can use:

```
ps -u $(whoami)
```

The whoami command returns the username you're logged in with, and **ps -u** lists all processes for that user.

LSOF Command : List Open Files.

List All Open Files
The simplest use of lsof is to run it without any options, which will display a list of all open files:

```
lsof
```

List Open Files by a Specific User

We can list all open files for a specific user using the -u option followed by the username:

```
lsof -u username
```

List Open Files by Process ID (PID)

To list files opened by a specific process, We can use the -p option followed by the process ID:

```
lsof -p [PID]
```

Find Processes Using a Specific Port
If We need to identify which process is using a specific port (let's say port 8080), you can do:

```
lsof -i :8080
```

Find Processes Using a Specific Protocol
We can also search by network protocol (TCP or UDP):

```
lsof -i tcp
```

List Open Files in a Directory

To list all processes that have opened files in a specific directory, We can use:

```
lsof +D /path/to/directory
```

Find Open Files by Application Name
We can find open files associated with a specific application (e.g., Chrome):

```
lsof -c Chrome
```

Kill a Process :

```
kill -9 [pid]
```

DF Command:

The df command in Linux is used to display information about the disk space usage of mounted filesystems. The command shows details such as the total size, available space, and used space on the disk partitions.

The basic syntax of the df command is:

```
df [OPTION]... [FILE]...
```

Options

- -h: Human-readable format, shows sizes in "K", "M", "G", etc.
- -T: Display the file system type.
- -a: Display all filesystems, including pseudo filesystems.
- -t: Filter by filesystem type.
- -x: Exclude a particular type of filesystem.

1. Basic Usage

The basic df command with no arguments displays disk usage in a tabular form:

```
df
```

2. Human-Readable Format
To get the sizes in a human-readable format:

df -h

3. Display File System Type
To display the type of filesystem (ext4, tmpfs, etc.):

df -T

4. Display Information About a Specific Filesystem or Directory
To display information about the filesystem containing a particular directory:

```
df /path/to/directory
```

5. Display Only Specific Filesystem Type
For instance, if you want to see only ext4 partitions:

```
df -t ext4
```

6. Exclude Specific Filesystem Type
If you want to exclude certain types, like tmpfs:

```
df -x tmpfs
```

7. All Filesystems Including Pseudo
To display all filesystems including the pseudo filesystems:

```
df -a
```

Practical Use-Cases :-

- **Checking Disk Space Before Download or Installation:** Before downloading a large file or installing a new application, you can use df -h to quickly check if enough disk space is available.

- **Monitoring Disk Usage in Scripts:** System administrators might want to include df in scripts that alert them when disk usage exceeds a certain threshold.

- **System Health Checks:** df can be a part of a series of commands you might run for regular system health checks, possibly alongside commands like top, free, etc.

- **Identifying Mount Points:** With the -T option, you can also determine what type of filesystem is used at each mount point, which can be useful for advanced storage configurations.

- **Excluding Irrelevant Entries:** In large deployments with network-mounted drives, RAM disks, etc., df with -x or -t options can filter out the noise, focusing on what is important.

- **Debugging Disk Full Issues:** If an application complains about a full disk, df is the quickest way to confirm whether you're actually out of space and on which filesystem.

- **Disk Cleanup:** By regularly checking disk usage, you can identify directories or partitions that are nearing capacity and can clean or archive files accordingly.