Herramientas de línea de comandos

Introducción a las herramientas de línea de comandos

¿Qué es una línea de comandos?

La línea de comandos (CLI, Command-Line Interface, por sus siglas en inglés) es una interfaz de usuario que permite interactuar con el sistema operativo mediante la introducción de comandos de texto. A diferencia de las interfaces gráficas (GUI), en las que el usuario interactúa principalmente a través del ratón y otros dispositivos de entrada visual, en la CLI el usuario ejecuta programas o tareas escribiendo comandos y recibiendo la salida de estos en formato de texto.

En sistemas operativos como GNU/Linux, la línea de comandos es un componente fundamental que permite a los usuarios y administradores realizar operaciones a nivel del sistema con mayor precisión y flexibilidad. A menudo se accede a la CLI a través de un programa llamado *terminal* o *consola*.

Ventajas del uso de la terminal frente a interfaces gráficas

Aunque las interfaces gráficas son más accesibles para los usuarios menos técnicos, la línea de comandos ofrece varias ventajas, especialmente para tareas avanzadas o repetitivas. Algunas de las principales ventajas son:

- **Eficiencia**: La CLI permite realizar operaciones complejas con unos pocos comandos. Por ejemplo, mover, copiar o renombrar múltiples archivos puede ser mucho más rápido en la línea de comandos que en una interfaz gráfica.
- **Control total**: La terminal ofrece acceso directo a casi todas las funciones del sistema, lo que da al usuario más control y flexibilidad.
- **Automatización**: Es más fácil automatizar tareas repetitivas mediante scripts en la terminal, algo que en una GUI podría requerir muchas acciones manuales.
- **Uso de recursos**: Las aplicaciones CLI generalmente consumen menos recursos del sistema en comparación con las aplicaciones gráficas, lo que las hace más eficientes en sistemas con hardware limitado.
- **Acceso remoto**: Mediante herramientas como SSH, se puede acceder a la línea de comandos de un sistema remoto de manera eficiente, lo cual es muy útil para la administración de servidores.
- **Compatibilidad universal**: Las herramientas de línea de comandos tienden a ser más uniformes y consistentes entre diferentes distribuciones o versiones del sistema operativo.

Shells disponibles en GNU/Linux

Un **intérprete de comandos**, o **shell**, es un programa cuya principal función es recibir las órdenes del usuario, interpretar esa cadena de texto, y luego ejecutar los programas correspondientes, actuando como intermediario entre el usuario y el sistema operativo. Su

función principal es interpretar los comandos escritos por el usuario, ejecutarlos, y mostrar el resultado.

En GNU/Linux existen varios shells, cada uno con características propias:

- **Bash (Bourne Again Shell)**: Es el shell más utilizado en distribuciones GNU/Linux, conocido por su versatilidad y amplia compatibilidad. Bash ofrece características avanzadas como historial de comandos, redirección, tuberías, y scripting.
- **Zsh (Z Shell)**: Similar a Bash pero con características adicionales que lo hacen más flexible y poderoso. Ofrece autocompletado avanzado, corrección automática de comandos mal escritos, y una mayor capacidad de personalización. Muchos usuarios lo prefieren por su facilidad de uso y extensibilidad.
- **Fish (Friendly Interactive Shell)**: Es conocido por ser fácil de usar desde el principio, con una experiencia más interactiva y moderna. Fish ofrece sugerencias automáticas de comandos basadas en el historial y tiene una sintaxis más simple para escribir scripts, pero puede no ser compatible con ciertos scripts de Bash.
- **Dash (Debian Almquist Shell)**: Es un shell ligero y rápido, enfocado en la ejecución de scripts POSIX. No ofrece tantas funcionalidades interactivas como Bash o Zsh, pero es excelente para scripts ligeros.
- **Tcsh**: Un shell que deriva de Csh (C Shell). Es popular por su sintaxis, similar a la del lenguaje de programación C, y es usado mayormente en entornos específicos como sistemas BSD. Aunque no tan popular en distribuciones Linux, algunos usuarios prefieren su estilo de scripting.

Cada shell tiene sus ventajas dependiendo del tipo de usuario y las necesidades del entorno. Mientras que Bash es el estándar de facto, otros shells como Zsh y Fish están ganando popularidad por ofrecer características avanzadas y mejoras en la experiencia del usuario.

El Shell Bash (Bourne Again Shell)

Bash es el shell por defecto en la mayoría de las distribuciones GNU/Linux y uno de los más populares y ampliamente utilizados. Desarrollado como una mejora sobre el Bourne Shell original (sh) de Unix, Bash no solo hereda todas sus características, sino que añade funcionalidades adicionales que lo hacen más potente y flexible.

Historia de Bash

Bash fue creado en 1989 por Brian Fox como parte del proyecto GNU con el objetivo de proporcionar una alternativa libre al shell Bourne de Unix. De ahí su nombre, **Bourne Again Shell**, que hace un juego de palabras con "born again" (nacer de nuevo). Desde entonces, Bash ha evolucionado hasta convertirse en un componente clave de la mayoría de los sistemas basados en Unix.

Características clave de Bash

Algunas de las características más importantes de Bash que lo diferencian de otros shells son:

- **Historial de comandos**: Bash mantiene un registro de los comandos ejecutados en sesiones anteriores. Los usuarios pueden navegar por este historial usando las teclas de flechas, lo que facilita la reutilización de comandos sin tener que reescribirlos.
- **Redirección de entrada/salida**: Al igual que otros shells, Bash permite redirigir la salida de los comandos a archivos o incluso a otros comandos. Esto es fundamental para la creación de flujos de trabajo eficientes.
- **Alias**: Bash permite crear alias para abreviar comandos largos o complejos. Por ejemplo, puedes crear un alias para `ls -la` con el nombre `ll`.
- **Variables**: Los usuarios pueden definir variables para almacenar información temporalmente dentro de una sesión de Bash, lo cual es extremadamente útil en scripts y automatización.
- **Scripting avanzado**: Bash no solo es un shell interactivo, sino también un potente lenguaje de scripting. Puedes escribir scripts complejos con soporte para variables, condicionales, bucles, y funciones.
- **Autocompletado**: Una de las características más útiles de Bash es el autocompletado. Al presionar *Tab*, Bash puede completar automáticamente comandos, nombres de archivos, y directorios.
- **Expansión de comodines (globbing)**: Bash soporta el uso de comodines como *****, **?**, y **[]** para trabajar con múltiples archivos al mismo tiempo.
- **Substitución de comandos**: Permite ejecutar un comando dentro de otro usando la sintaxis `$(comando)` o `comando`. Esto es útil para usar la salida de un comando como entrada para otro.
- **Shell scripting avanzado**: Además de ejecutar comandos, Bash permite escribir scripts con funcionalidades avanzadas como bucles (*for*, *while*), condicionales (*if*, *case*), y funciones.

Navegación en el historial y reutilización de comandos

Bash incluye un historial que facilita la reutilización de comandos previos. Este historial se almacena en el archivo oculto `~/.bash_history`. Las teclas de flechas *Arriba* y *Abajo* permiten desplazarse por los comandos ya ejecutados. Además, existen atajos útiles, como:

- `Ctrl + r`: Busca de manera interactiva un comando en el historial.
- `!comando`: Repite el último comando que comience con la palabra comando.
- `!!`: Ejecuta el último comando ingresado.

Configuración y personalización de Bash

El shell Bash es altamente configurable y personalizable. Algunas formas en las que los usuarios pueden ajustar Bash a sus preferencias incluyen:

- **Archivos de configuración**: Bash tiene varios archivos de configuración, como `.bashrc` y `.bash_profile`, que permiten a los usuarios definir alias, variables de entorno, y personalizar su entorno de trabajo.
    - **`~/.bashrc`**: Este archivo de configuración se ejecuta cada vez que abres una nueva terminal. Puedes agregar alias, funciones y otras personalizaciones aquí.
    - **`~/.bash_profile`**: Usado cuando inicias sesión en Bash, generalmente contiene configuraciones para sesiones de login.
- **Alias**: Los usuarios pueden crear alias personalizados para ejecutar comandos largos con nombres más cortos. Por ejemplo:

  ```
  alias ll='ls -la'
  ```

  Esto permite ejecutar `ll` en lugar de escribir `ls -la` cada vez.
- **Prompt personalizado**: Bash permite personalizar el prompt, es decir, la cadena de texto que se muestra antes de que el usuario escriba un comando. El prompt se configura a través de la variable de entorno PS1. Un ejemplo básico de un prompt personalizado es:

  ```
  export PS1="\u@\h:\w\$ "
  ```

  Este prompt muestra el nombre de usuario *(\u)*, el nombre del host *(\h)*, el directorio actual *(\w)*, seguido del símbolo *$*.

Tareas avanzadas en Bash

- **Control de procesos**: Bash permite ejecutar comandos en segundo plano usando el operador & y luego gestionarlos con los comandos `jobs`, `fg` (foreground), y `bg` (background). Además, es posible suspender y reanudar procesos con `Ctrl + Z` y `fg`.
- **Redirección y tuberías**: El manejo de entrada/salida en Bash es esencial para automatizar tareas complejas. La redirección permite enviar la salida de un comando a un archivo o tomar la entrada de un archivo, mientras que las tuberías permiten encadenar comandos.
    - Redirección de salida:

      ```
      comando > archivo.txt
      ```

    - Tubería para pasar la salida de un comando a otro:

      ```
      comando1 | comando2
      ```

  Exploraremos estas utilidades en detalle más adelante.

Seguridad en Bash

Bash también es crucial en términos de seguridad en sistemas GNU/Linux. Algunas de las mejores prácticas para usar Bash de manera segura incluyen:

- **Manejo de permisos**: Bash permite gestionar permisos de archivos con comandos como `chmod`, y `chown`. Asegurarse de que los archivos y scripts tienen los permisos correctos es clave para la seguridad del sistema.
- **Control de usuarios**: En Bash, los usuarios pueden cambiar de usuario utilizando el comando `su` o elevar privilegios con `sudo`, permitiendo realizar tareas administrativas de manera segura sin comprometer la seguridad del sistema.
- **Scripting seguro**: Cuando se escriben scripts en Bash, es importante usar buenas prácticas de programación como validar entradas de usuario, evitar el uso de contraseñas en texto plano y asegurarse de que el script no ejecuta comandos no verificados.

Estructura general de un comando

Cuando trabajamos en la línea de comandos de un sistema GNU/Linux, cada comando que ejecutamos tiene una estructura básica, que puede variar en complejidad según las necesidades. Entender esta estructura es fundamental para trabajar de manera efectiva con la terminal.

Sintaxis de los comandos

La sintaxis básica de un comando en la línea de comandos generalmente sigue este formato:

```
comando [opciones] [argumentos]
```

donde:

- comando: Es el nombre del programa o instrucción que deseas ejecutar. Por ejemplo, `ls`, `cp`, `mkdir`, etc.
- opciones: Son parámetros que modifican el comportamiento del comando. Las opciones usualmente se anteceden con un guion simple (`-`) o doble (`--`) para diferenciarlas de los argumentos.
    - Ejemplo con guion simple: `ls -l`
    - Ejemplo con guion doble: `ls --long`
- argumentos: Son los valores que el comando necesita para ejecutarse sobre un archivo, directorio o dato específico. Por ejemplo, al copiar un archivo, el archivo de origen y el destino serían los argumentos.

Vamos a desglosar cada componente con un ejemplo práctico:

```
cp -r archivo.txt /ruta/de/destino/
```

1. **Comando**: `cp`
    - Este es el programa que se ejecutará. En este caso, `cp` es el comando utilizado para copiar archivos y directorios.

2. **Opciones**: `-r`
    - Las opciones modifican el comportamiento del comando. En este ejemplo, `-r` (o `--recursive`) le indica a `cp` que copie directorios de manera recursiva.
3. **Argumentos**: `archivo.txt` y `/ruta/de/destino/`
    - Los argumentos proporcionan los datos sobre los que opera el comando. Aquí, `archivo.txt` es el archivo que será copiado, y `/ruta/de/destino/` es el lugar donde se copiará el archivo.

Un comando puede tener múltiples opciones y argumentos, pero no todos los comandos requieren ambos. Algunos comandos se pueden ejecutar sin argumentos u opciones, como:

```
pwd
```

Este comando simplemente imprime en la salida estándar el directorio de trabajo actual sin requerir opciones o argumentos.

Ejecución de comandos secuenciales y paralelos

En la línea de comandos, se pueden ejecutar múltiples comandos de manera secuencial o en paralelo, lo que ofrece flexibilidad al usuario para automatizar tareas.

- **Ejecución secuencial**: Se ejecutan comandos uno tras otro, en el orden en que aparecen. Para ejecutar comandos secuencialmente, se puede usar el punto y coma (`;`):

  ```
  comando1 ; comando2 ; comando3
  ```

  Por ejemplo:

  ```
  mkdir nuevo_directorio ; cd nuevo_directorio ; touch archivo.txt
  ```

  Aquí, se crean secuencialmente un nuevo directorio, se cambia a ese directorio, y luego se crea un archivo en el mismo.

- **Ejecución paralela**: Los comandos se pueden ejecutar en paralelo usando el operador &. Esto es útil cuando los comandos no dependen entre sí y pueden ejecutarse simultáneamente.

  ```
  comando1 & comando2 &
  ```

  Por ejemplo:

  ```
  ping google.com & ping bing.com &
  ```

  Esto ejecuta ambos comandos `ping` de manera simultánea, cada uno en su propio proceso.

Variables de entorno y su influencia en la ejecución de comandos

Las **variables de entorno** son un conjunto de variables del sistema que influyen en el comportamiento de los comandos en la terminal. Se almacenan en la memoria y contienen

información que los programas y procesos utilizan para funcionar correctamente. Algunas variables importantes son:

- **PATH**: Define los directorios en los que el sistema busca comandos ejecutables. Si un comando no está en uno de estos directorios, la terminal no podrá ejecutarlo directamente. Por ejemplo, cuando escribes `ls`, el sistema busca el comando `ls` en los directorios listados en PATH.
  Puedes ver el valor de PATH ejecutando:

  ```
  echo $PATH
  ```

  Si quieres agregar un nuevo directorio al PATH, puedes hacerlo temporalmente con:

  ```
  export PATH=$PATH:/nuevo/directorio
  ```

- **HOME**: Define el directorio personal del usuario actual. Los comandos que se refieren a archivos o directorios dentro del entorno del usuario pueden utilizar esta variable para apuntar directamente al directorio personal sin tener que escribir la ruta completa.
  Ver el valor de HOME:

  ```
  echo $HOME
  ```

- **USER**: Contiene el nombre de usuario actual en la sesión de la terminal. Esto es útil para scripts que requieren identificar al usuario que ejecuta el script.
- **PS1**: Define el aspecto del **prompt**, la línea que aparece cada vez que el terminal está esperando un comando del usuario. El PS1 se puede personalizar para mostrar información como el directorio actual, el nombre del usuario, o incluso la hora.
- **SHELL**: Indica qué shell se está utilizando en la sesión actual.

Las variables de entorno se pueden establecer o modificar en tiempo real dentro de la sesión actual con el comando `export`:

```
export NOMBRE="valor"
```

Por ejemplo:

```
export EDITOR=nano
```

Esto indica que cualquier programa que consulte qué editor de texto utilizar debe usar `nano`.

Algunas variables de entorno, como PATH, se pueden modificar para que los cambios persistan entre sesiones. Para ello, puedes agregarlas a los archivos de configuración de Bash, como `.bashrc` o `.bash_profile`.

Para listar todas las variables de entorno que tienes definidas en un sistema GNU/Linux, se puede utilizar el comando `printenv`. Este comando muestra una lista completa de las variables de entorno activas junto con sus respectivos valores. La sintaxis básica es:

```
printenv
```

Si se desea consultar el valor de una variable de entorno específica, se puede proporcionar su nombre como argumento:

```
printenv NOMBRE_VARIABLE
```

Por ejemplo, para ver el valor de la variable PATH, se puede ejecutar:

```
printenv PATH
```

Otra alternativa para listar las variables de entorno es el comando `env`, que proporciona una salida similar:

```
env
```

Ambos comandos son útiles para verificar las variables que están influyendo en el comportamiento del sistema y sus comandos en la terminal.

Comandos básicos

Los **comandos básicos** son el conjunto de instrucciones fundamentales que todo usuario de Linux debe conocer para navegar por el sistema de archivos, gestionar archivos y directorios, obtener información del sistema y manejar permisos. Estos comandos son esenciales tanto para usuarios novatos como para administradores de sistemas, ya que proporcionan un control directo y flexible sobre el entorno de trabajo.

En esta sección, aprenderemos los comandos más utilizados en Linux para movernos dentro del sistema, realizar operaciones sobre archivos y directorios, consultar el estado del sistema, y gestionar usuarios y permisos.

Navegación por el sistema de archivos

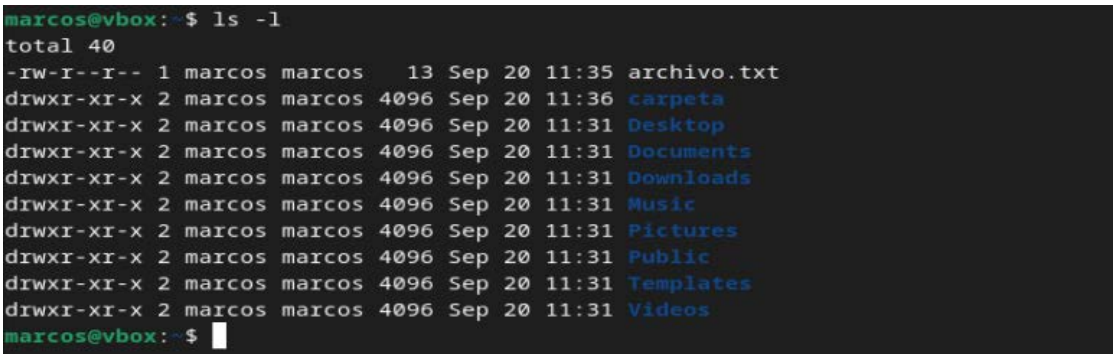
En la línea de comandos de GNU/Linux, uno de los primeros conceptos que debemos dominar es la navegación por el sistema de archivos. Los siguientes comandos son esenciales para movernos por los directorios y archivos.

*Comando **`ls`***

El comando `ls` es uno de los más utilizados en la terminal de GNU/Linux. Sirve para listar el contenido de un directorio, es decir, los archivos y subdirectorios que se encuentran en un directorio específico. Cuando se ejecuta sin ningún argumento, este comando mostrará una lista de los archivos y directorios **visibles** en el directorio actual. La salida será una lista en columnas que solo incluye los nombres de los archivos y carpetas. Un ejemplo básico de salida podría ser:

```
marcos@vbox:~$ ls
archivo.txt  Desktop    Downloads  Pictures  Templates
carpeta      Documents  Music      Public    Videos
marcos@vbox:~$
```

Cuando se ejecuta con la opción `-l`, `ls` muestra un listado más detallado, con información adicional sobre cada archivo o directorio. La salida de este comando se ve así:



Cada columna de esta salida tiene un significado específico:

1. **Cantidad de bloques total** (`total 40`):
    - La frase `total 40` que aparece al principio del listado indica el **tamaño total en bloques de disco** de los archivos y directorios que se están listando en el directorio actual. Esto puede incluir archivos, directorios vacíos, enlaces simbólicos, etc.
2. **Permisos de archivo** (`-rw-r--r--`):
    - El primer carácter indica el tipo de archivo:
        - -: archivo normal
        - d: directorio
        - l: enlace simbólico
    - Los siguientes nueve caracteres se dividen en tres grupos de tres, que representan los permisos para:
        - **Propietario**: permisos del usuario que es dueño del archivo.
        - **Grupo**: permisos del grupo asociado al archivo.
        - **Otros**: permisos para todos los demás usuarios del sistema.
    - Por ejemplo, para `archivo.txt`, `-rw-r--r--` indica que el propietario puede leer y escribir (`rw-`), el grupo sólo puede leer (`r--`), y los demás también sólo pueden leer (`r--`).
3. **Número de enlaces** (1):
    - Esta columna indica el número de enlaces duros al archivo o directorio. En el caso de los directorios, indica cuántos subdirectorios o enlaces simbólicos apuntan a él.
4. **Propietario** (usuario):
    - Muestra el nombre del usuario que posee el archivo o directorio.
5. **Grupo** (grupo):
    - Indica el grupo al que pertenece el archivo o directorio.

6. **Tamaño del archivo** (1024):
    - El tamaño del archivo en bytes. Para directorios, este valor refleja el tamaño del bloque que contiene la información sobre el contenido del directorio, no el tamaño total de los archivos dentro de él.
7. **Fecha y hora de la última modificación** (Sep 15 14:20):
    - La fecha y hora en que el archivo o directorio fue modificado por última vez.
8. **Nombre del archivo o directorio** (archivo.txt o carpeta):
    - El nombre del archivo o directorio.

Otra opción comúnmente utilizada es `-a`. esta muestra todos los archivos, incluidos los **archivos ocultos** (aquellos cuyo nombre comienza con un punto .). Un archivo oculto común es `.bashrc`, que contiene configuraciones de Bash.

```
marcos@Casa-PC:~$ ls -a
.       archivo1.txt  archivo4.txt  .bash_history  fibonacci.c    fibonacci.m  fibo.ocpp  .sudo_as_admin_successful
..      archivo2.txt  archivo5.txt  .bash_logout   fibonacci.cpp  fibonacci.o  .local     tp3_ejemplos.tar
a.out   archivo3.txt  archivos.sh   .bashrc        fibonacci_cpp  fibonacci.s  .profile
```

La opción `-h`, cuando se combina con `-l`, esta opción muestra los tamaños de los archivos en un formato más legible para los humanos, como KB, MB o GB en lugar de solo bytes.

```
marcos@Casa-PC:~$ ls -lh
total 84K
-rwxr-xr-x 1 marcos marcos  17K Sep 16 02:11 a.out
-rw-r--r-- 1 marcos marcos    0 Sep 16 00:51 archivo1.txt
-rw-r--r-- 1 marcos marcos    0 Sep 16 00:51 archivo2.txt
-rw-r--r-- 1 marcos marcos    0 Sep 16 00:51 archivo3.txt
-rw-r--r-- 1 marcos marcos    0 Sep 16 00:51 archivo4.txt
-rw-r--r-- 1 marcos marcos    0 Sep 16 00:51 archivo5.txt
-rwxr-xr-x 1 marcos marcos  348 Sep 16 00:50 archivos.sh
-rw-r--r-- 1 marcos marcos  726 Sep 16 01:58 fibonacci.c
-rw-r--r-- 1 marcos marcos  794 Sep 16 02:02 fibonacci.cpp
-rwxr-xr-x 1 marcos marcos  17K Sep 16 02:06 fibonacci_cpp
-rw-r--r-- 1 marcos marcos  802 Sep 16 02:21 fibonacci.m
-rw-r--r-- 1 marcos marcos 2.2K Sep 16 02:09 fibonacci.o
-rw-r--r-- 1 marcos marcos 7.4K Sep 16 02:33 fibonacci.s
-rw-r--r-- 1 marcos marcos 3.8K Sep 16 02:08 fibo.ocpp
-rw-r--r-- 1 marcos marcos  10K Sep 16 00:51 tp3_ejemplos.tar
marcos@Casa-PC:~$
```

Una opción que puede ser útil es `-R`, que lista los archivos y directorios de manera **recursiva**, es decir, mostrará no solo el contenido del directorio actual, sino también el de todos los subdirectorios que contiene. Esto mostrará una lista detallada de cada subdirectorio, por ejemplo:

```
marcos@Casa-PC:/home$ ls -R
.:
marcos

./marcos:
a.out         archivo2.txt  archivo4.txt  archivos.sh  fibonacci.cpp  fibonacci.m  fibonacci.s  tp3_ejemplos.tar
archivo1.txt  archivo3.txt  archivo5.txt  fibonacci.c  fibonacci_cpp  fibonacci.o  fibo.ocpp
marcos@Casa-PC:/home$
```

En muchas configuraciones, `ls` se utiliza con la opción `--color=auto` de manera predeterminada, lo que proporciona una salida con colores para diferenciar entre tipos de archivos (por ejemplo, archivos ejecutables en verde, directorios en azul).

*Comando **tree***

Un comando complementario al comando `ls` es el comando `tree`, el cual muestra los directorios y archivos como un árbol:

Esto puede ser muy útil para tener un vistazo general de un directorio en particular.

*Comando **cd***

El comando `cd` (change directory) es utilizado para cambiar de directorio en la línea de comandos de GNU/Linux. Este comando es fundamental para la navegación en el sistema de archivos. A continuación, vamos a profundizar en cómo funciona, las variables de entorno que lo afectan, y el uso de caracteres especiales para simplificar la navegación.

La sintaxis básica de `cd` es la siguiente:

```
cd /ruta/a/directorio
```

Este comando cambiará el directorio actual a la ruta especificada. Si no se especifica ninguna ruta, `cd` nos llevará automáticamente al directorio personal del usuario actual.

Existen varias variables de entorno que influyen directamente en cómo funciona el comando `cd`. Las más importantes son:

- **HOME**: Esta variable contiene la ruta al directorio personal del usuario. Es el directorio al que se accede cuando se ejecuta `cd` sin argumentos. Por ejemplo, si el usuario actual es `juan` y su directorio personal es `/home/juan`, ejecutar simplemente `cd` nos llevará a `/home/juan`.
- **OLDPWD**: Esta variable almacena la ruta del último directorio en el que nos encontrábamos antes de cambiar al actual. Usando `cd -`, podemos regresar al directorio anterior fácilmente, esto es, cambiará entre el directorio actual y el último que visitamos de manera alternada.

- **CDPATH**: Esta variable especifica una lista de directorios que cd debería buscar si el usuario proporciona una ruta relativa. Si el directorio no está en el directorio de trabajo actual, el shell buscará en las rutas definidas en CDPATH. Por ejemplo, si definimos CDPATH la entrada `export CDPATH=/home/juan/proyectos`, y luego ejecutamos `cd mi_proyecto`, entonces el comando `cd` buscará dentro de `/home/juan/proyectos/mi_proyecto` aunque no estemos actualmente en ese directorio.

Existen varios **caracteres especiales** que son útiles al usar el comando cd para navegar más eficientemente en el sistema de archivos:

- **Tilde (~)**: La tilde es un atajo para el **directorio personal** del usuario. En lugar de escribir la ruta completa al directorio personal, podemos usar ~:

  ```
  cd ~
  ```

  Esto nos llevará al mismo lugar que `cd` sin argumentos. También podemos combinarla con otras rutas:

  ```
  cd ~/Documentos
  ```

  Esto nos llevará al subdirectorio `Documentos` dentro del directorio personal.

- **Punto (.)**: El punto hace referencia al **directorio actual**. Aunque no se usa mucho con `cd`, es útil en otros comandos para referirse al directorio en el que nos encontramos actualmente.

  ```
  cd .
  ```

  No cambiará de directorio, ya que . significa "directorio actual".

- **Doble punto (..)**: Los dos puntos (..) hacen referencia al directorio padre o superior. Usarlo con `cd` nos permite **subir un nivel** en la jerarquía de directorios.

  ```
  cd ..
  ```

  Si estamos en el directorio `/home/juan/Documentos`, este comando nos llevará a `/home/juan`. También podemos encadenar .. para subir más niveles:

  ```
  cd ../../
  ```

  Esto nos llevará dos niveles arriba.

- **Guion (-)**: El guion es un atajo para regresar al **último directorio** en el que nos encontrábamos. Al ejecutar `cd -`, cambiaremos al último directorio usado antes del actual.

  ```
  cd -
  ```

  Si estábamos en `/home/juan/Documentos` y cambiamos a `/var/log`, ejecutar `cd -` nos regresará a `/home/juan/Documentos`.

- **Barra diagonal (/)**: La barra diagonal es el símbolo que se utiliza para separar los directorios en una ruta. También se utiliza como referencia al directorio raíz del sistema, el directorio más alto en la jerarquía del sistema de archivos.

- **Ruta absoluta**: Si empezamos una ruta con /, estamos indicando una ruta **absoluta**, es decir, una ruta completa desde el directorio raíz.

```
cd /home/juan/Documentos
```

- **Ruta relativa**: Si la ruta no comienza con /, estamos indicando una **ruta relativa** al directorio en el que nos encontramos actualmente.

```
cd Documentos
```

Esto nos llevará al directorio `Documentos` dentro del directorio actual.

*Comando **pwd***

Dentro de los comandos para la navegación por el sistema de archivos, uno muy útil es el comando `pwd`. Este imprime en pantalla la ruta del directorio de trabajo actual:

```
marcos@Casa-PC:~$ pwd
/home/marcos
marcos@Casa-PC:~$
```

No confundir el directorio de trabajo por defecto con el directorio de trabajo actual, ya que el directorio de trabajo actual es el último al que accedimos con un comando `cd`.

Gestión de archivos y directorios

En GNU/Linux, los archivos y directorios son los bloques fundamentales de almacenamiento y organización de datos. La terminal permite realizar operaciones básicas y avanzadas sobre ellos de manera eficiente. A través de comandos simples como `cp`, `mv`, `rm` y `mkdir`, podemos copiar, mover, renombrar, eliminar archivos y directorios, así como crear nuevas estructuras de almacenamiento. En esta sección, exploraremos cómo gestionar archivos y directorios de manera efectiva utilizando comandos básicos, optimizando el flujo de trabajo y la organización en el sistema de archivos.

*Comando **cp***

El comando `cp` (copy) en GNU/Linux se utiliza para copiar archivos y directorios de un lugar a otro. Podemos copiar archivos individuales, múltiples archivos o incluso estructuras completas de directorios. A continuación, veremos la sintaxis básica y algunas de las opciones más comunes que mejoran su funcionalidad.

La sintaxis más simple del comando `cp` es:

```
cp archivo_origen archivo_destino
```

Este comando copia el archivo de origen (`archivo_origen`) al destino (`archivo_destino`). Si `archivo_destino` ya existe, será sobrescrito sin previo aviso, a menos que utilicemos alguna opción que solicite confirmación.

Para copiar un **directorio completo** y su contenido, necesitamos usar la opción `-r` (recursiva):

```
cp -r directorio_origen directorio_destino
```

Esta opción asegura que todos los archivos y subdirectorios dentro del directorio de origen se copien al destino. Sin la opción `-r`, `cp` no puede copiar directorios.

Además de `-r`, existen varias otras opciones útiles para el comando `cp` que nos permiten tener mayor control sobre el proceso de copiado. La opción `-i` (interactivo) solicita confirmación antes de sobrescribir archivos existentes:

```
cp -i archivo_origen archivo_destino
```

Si archivo_destino ya existe, cp pedirá confirmación antes de sobrescribirlo:

```
cp: overwrite 'archivo_destino'? y/n
```

La opción `-u` (actualizar): Copia el archivo **solo si el archivo de destino es más antiguo** que el de origen o si no existe.

```
cp -u archivo_origen archivo_destino
```

Esta opción es útil para **sincronizar directorios** y evitar sobrescribir archivos más recientes en el destino.

Si utilizamos la opción `-v` (verbose), esta muestra en pantalla una descripción detallada de lo que está haciendo el comando, es decir, muestra cada archivo o directorio que se está copiando:

```
cp -v archivo_origen archivo_destino
```

La salida sería algo como:

```
'archivo_origen' -> 'archivo_destino'
```

Con la opción `-p` (preservar atributos) se copia el archivo manteniendo sus atributos originales, como los permisos, el propietario, el grupo y las marcas de tiempo de modificación.

cp -p archivo_origen archivo_destino

Esto asegura que el archivo copiado tenga los mismos metadatos que el archivo original, útil en situaciones donde los permisos y la propiedad del archivo son críticos.

Al usar `-a` (archivado), esta opción combina varias opciones (`-r`, `-p`, y `-d`) para hacer una copia exacta de un directorio, manteniendo la estructura de archivos, permisos, enlaces simbólicos y otros atributos. Es ideal para crear **copias de seguridad**.

```
cp -a directorio_origen directorio_destino
```

El uso de `-a` asegura que el directorio copiado mantenga todas sus propiedades, incluyendo enlaces simbólicos y metadatos.

Con `–backup`, esta opción crea una copia de seguridad del archivo de destino si ya existe, agregando un sufijo (~) al archivo original en el destino.

```
cp --backup archivo_origen archivo_destino
```

Si `archivo_destino` ya existe, `cp` lo renombrará como `archivo_destino~` antes de copiar el nuevo archivo.

Si utilizamos `--parents`, esta opción copia los archivos junto con su estructura completa de directorios. Esto es útil cuando queremos copiar un archivo específico, pero también mantener la estructura de directorios desde la raíz.

```
cp --parents /ruta/completa/al/archivo /destino/
```

Si la ruta de origen es `/home/juan/documentos/archivo.txt` y queremos copiar `archivo.txt` con toda su estructura de directorios, el comando creará en el destino la misma estructura `/home/juan/documentos/`.

*Comando **mv***

Este comando se utiliza esencialmente para mover archivos o directorios de una ubicación a otra. La sintaxis es similar a la de `cp`, sin embargo, hay una diferencia fundamental entre estos dos comandos. Se puede pensar en `cp` como una operación de copiar y pegar, mientras que `mv` puede equipararse a cortar y pegar.

Cuando se utiliza este comando, el archivo o directorio se mueve a un nuevo lugar, y el original deja de existir. Por ejemplo:

```
mv archivo_fuente directorio_destino
```

moverá el `archivo_fuente` lo pondrá en el `directorio_destino`. Si lo utilizamos con más de un archivo, por ejemplo:

```
mv archivo1.txt archivo2.txt archivo3.txt directorio_destino
```

moverá todos los archivos al directorio destino.

Otro uso común de este comando es para renombrar archivos, utilizando como destino un nombre de archivo en lugar de un directorio:

```
mv archivo_origen archivo_objetivo
```

renombrará el archivo original con el nombre del archivo destino. Si no existe el `archivo_objetivo`, se creará, pero si ya existe lo sobrescribirá, por lo que se debe tener especial cuidad al realizar esta operación.

También es posible mover directorios, utilizando la misma sintaxis que con archivos, pero especificando directorios como argumentos. Si el directorio de destino existe, todo el directorio origen se moverá dentro del directorio destino, lo que significa que se convertirá en un subdirectorio del directorio destino.

Con la opción `-n` se evita la sobreescritura del archivo en caso de que exista en el destino, mientras que la opción `-i` pregunta si se quiere sobrescribir. También es posible realizar un backup del archivo sobrescrito con la opción `-b`, esto es, se sobrescribe el archivo y se guarda una copia con el mismo nombre, pero con el carácter ~ al final. Se puede especificar el sufijo que se utilizará para el backup:

```
mv -S .back -b archivo.txt directorio_objetivo/archivo.txt
```

Aquí, con la opción `-S`, el archivo destino tomará el nombre `archivo.txt.back`.

*Comando **rm***

Con el comando `rm` es posible eliminar archivos. Esta eliminación es permanente, por lo que se debe ser cuidadoso en su uso. La sintaxis es simplemente:

```
rm archivo
```

Para evitar eliminaciones involuntarias es posible utilizar la opción `-i`, la cual preguntará antes de eliminar:

También es posible forzar la eliminación con la opción `-f`; esto es útil cuando es quiere eliminar archivos sin mensajes de advertencia, típicamente cuando se quiere hacer un vaciado rápido de un directorio.

Con la opción `-d` es posible eliminar directorios, siempre y cuando esté vacío. Si el directorio no está vacío, se utiliza la opción `-r`.

*Comando **mkdir***

En GNU/Linux se utiliza el comando `mkdir` (make directory) para crear nuevos directorios. Es una de las operaciones más básicas y esenciales para organizar archivos en un sistema de archivos. La sintaxis básica es:

```
mkdir nombre_del_directorio
```

Este comando crea un directorio con el nombre especificado en el directorio actual. Si ya existe un directorio con ese nombre, se mostrará un mensaje de error indicando que el archivo o directorio ya existe.

Una funcionalidad interesante de `mkdir` es el uso de **llaves** para crear varios directorios al mismo tiempo con una sola línea de comando. Esto es útil cuando se desea crear una estructura compleja de directorios o directorios con nombres similares. Por ejemplo, si se quiere crear varios directorios de nombre secuencial o con un patrón similar:

```
mkdir directorio{1,2,3}
```

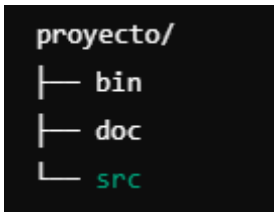
Este comando creará tres directorios:

```
directorio1  directorio2  directorio3
```

También es posible anidar el uso de llaves para crear subdirectorios de manera más eficiente:

```
mkdir -p proyecto/{src,bin,doc}
```

Esto creará la siguiente estructura:



Una opción extremadamente útil es `-p` (parents), cuando se quiere crear una estructura de directorios anidada sin necesidad de crear cada nivel manualmente. Si los directorios intermedios no existen, `mkdir -p` los crea automáticamente.

```
mkdir -p /ruta/a/nuevo/directorio
```

En este caso, si los directorios `/ruta/a/nuevo` no existen, el comando los creará todos de una vez. Sin la opción `-p`, se recibiría un error si se intentara crear un directorio sin que existieran los directorios intermedios.

*Comando **rmdir***

El comando `rmdir` se utiliza para **eliminar directorios vacíos** en un sistema GNU/Linux. A diferencia del comando `rm`, que puede eliminar archivos y directorios (incluso si contienen archivos), `rmdir` solo elimina directorios que no contienen ningún archivo o subdirectorio. Es útil cuando estamos seguros de que el directorio que queremos eliminar está vacío y no requiere opciones adicionales para manejar archivos dentro de él.

La sintaxis más simple para eliminar un directorio vacío es:

```
rmdir nombre_del_directorio
```

Si el directorio no está vacío, `rmdir` no podrá eliminarlo y mostrará un mensaje de error. Esto lo diferencia del comando `rm -r`, que puede eliminar directorios completos con todo su contenido.

Aunque `rmdir` es un comando bastante sencillo, existen algunas opciones útiles que nos ayudan a gestionar la eliminación de directorios en diferentes escenarios.

- **`-p` (padres)**: Esta opción elimina no solo el directorio especificado, sino también sus directorios padres si también están vacíos. Es útil para limpiar múltiples niveles de directorios vacíos en una sola operación.

  ```
  rmdir -p ruta/a/directorio
  ```

  Si tanto `directorio`, `a`, como `ruta` están vacíos, este comando eliminará los tres directorios en un solo paso.

- **`--ignore-fail-on-non-empty`**: Esta opción le indica a `rmdir` que ignore los errores si el directorio no está vacío, en lugar de interrumpir el proceso y mostrar un mensaje de error. Aunque el directorio no será eliminado si contiene archivos, la operación continuará sin detenerse por este motivo.

  ```
  rmdir --ignore-fail-on-non-empty nombre_del_directorio
  ```

  Esto es útil si estamos ejecutando un script o comando que elimina múltiples directorios, algunos de los cuales podrían no estar vacíos, y no queremos que el proceso se detenga por un error.

Información del sistema

Los siguientes comandos proporcionan información útil sobre el sistema y sus recursos:

*Comando **df***

Muestra el uso del espacio en disco. Su sintaxis básica es:

```
marcos@Casa-PC:~$ df
Filesystem      1K-blocks      Used  Available Use% Mounted on
none              8172716         0    8172716   0% /usr/lib/modules/5.15.153.1-microsoft-standard-WSL2
none              8172716         4    8172712   1% /mnt/wsl
drivers         487528444  93074984  394453460  20% /usr/lib/wsl/drivers
/dev/sdc       1055762868    781088 1001278308   1% /
none              8172716        76    8172640   1% /mnt/wslg
none              8172716         0    8172716   0% /usr/lib/wsl/lib
rootfs            8169300      2208    8167092   1% /init
none              8169300         0    8169300   0% /dev
none              8172716         4    8172712   1% /run
none              8172716         0    8172716   0% /run/lock
none              8172716         0    8172716   0% /run/shm
none              8172716         0    8172716   0% /run/user
tmpfs             8172716         0    8172716   0% /sys/fs/cgroup
none              8172716        92    8172624   1% /mnt/wslg/versions.txt
none              8172716        92    8172624   1% /mnt/wslg/doc
C:\             487528444  93074984  394453460  20% /mnt/c
D:\            1953497084 326213700 1627283384  17% /mnt/d
E:\             468833276 238260012  230573264  51% /mnt/e
F:\             468833276 282801932  186031344  61% /mnt/f
marcos@Casa-PC:~$
```

Para mostrar el tamaño de las unidades en un formato legible (KB, MB, GB), se utiliza la opción `-h`:

```
marcos@Casa-PC:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
none            7.8G     0  7.8G   0% /usr/lib/modules/5.15.153.1-microsoft-standard-WSL2
none            7.8G  4.0K  7.8G   1% /mnt/wsl
drivers         465G   89G  377G  20% /usr/lib/wsl/drivers
/dev/sdc       1007G  763M  955G   1% /
none            7.8G   76K  7.8G   1% /mnt/wslg
none            7.8G     0  7.8G   0% /usr/lib/wsl/lib
rootfs          7.8G  2.2M  7.8G   1% /init
none            7.8G     0  7.8G   0% /dev
none            7.8G  4.0K  7.8G   1% /run
none            7.8G     0  7.8G   0% /run/lock
none            7.8G     0  7.8G   0% /run/shm
none            7.8G     0  7.8G   0% /run/user
tmpfs           7.8G     0  7.8G   0% /sys/fs/cgroup
none            7.8G   92K  7.8G   1% /mnt/wslg/versions.txt
none            7.8G   92K  7.8G   1% /mnt/wslg/doc
C:\             465G   89G  377G  20% /mnt/c
D:\             1.9T  312G  1.6T  17% /mnt/d
E:\             448G  228G  220G  51% /mnt/e
F:\             448G  270G  178G  61% /mnt/f
marcos@Casa-PC:~$
```

*Comando **du***

Muestra el uso de espacio en disco por archivos y directorios:

```
marcos@Casa-PC:~$ du
4       ./.local/share/nano
8       ./.local/share
12      ./.local
116     .
```

Para mostrar el tamaño total de un directorio en un formato legible:

```
marcos@Casa-PC:~$ du -sh /home/marcos/
116K    /home/marcos/
marcos@Casa-PC:~$
```

*Comando **free***

Muestra la cantidad de memoria disponible y utilizada en el sistema. La sintaxis básica:

```
marcos@Casa-PC:~$ free
               total        used        free      shared  buff/cache   available
Mem:        16345432      647044    15691644        2552      270080    15698388
Swap:        4194304           0     4194304
marcos@Casa-PC:~$
```

Para un formato más comprensible:

```
marcos@Casa-PC:~$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi       584Mi        15Gi       2.5Mi        97Mi        15Gi
Swap:          4.0Gi          0B       4.0Gi
marcos@Casa-PC:~$
```

*Comando **uname***

Muestra información sobre el sistema operativo. Su sintaxis básica es:

```
marcos@Casa-PC:~$ uname
Linux
```

Para obtener detalles más completos:

```
marcos@Casa-PC:~$ uname -a
Linux Casa-PC 5.15.153.1-microsoft-standard-WSL2 #1 SMP Fri Mar 29 23:14:13 UTC 2024 x86_64 GNU/Linux
marcos@Casa-PC:~$
```

Manejo de permisos y usuarios

En GNU/Linux, cada archivo y directorio tiene permisos asociados para controlar quién puede leer, escribir o ejecutar el archivo. Estos comandos son fundamentales para gestionar los permisos y propiedades de los archivos.

*Comando **chmod***

El comando `chmod` (change mode) se utiliza para **cambiar los permisos** de archivos y directorios en GNU/Linux. Los permisos controlan quién puede leer, escribir o ejecutar un archivo o directorio, y se pueden especificar de dos formas: en formato numérico o formato simbólico.

Cada archivo o directorio en GNU/Linux tiene tres conjuntos de permisos:

1. **Propietario**: El usuario que posee el archivo. No necesariamente coincide con quien creó el archivo.
2. **Grupo**: El grupo al que pertenece el archivo.
3. **Otros**: Todos los demás usuarios que no son ni el propietario ni parte del grupo.

Para cada uno de estos conjuntos de usuarios, existen tres tipos de permisos:

- **r** (read): Permiso para leer el archivo o listar el contenido de un directorio.

- **w** (write): Permiso para modificar el archivo o agregar/eliminar archivos en un directorio.
- **x** (execute): Permiso para ejecutar el archivo (si es un programa o script) o acceder a un directorio.

El formato numérico es una forma compacta de representar los permisos mediante tres números octales (0-7), uno para cada conjunto de usuarios (propietario, grupo y otros). Cada número es la suma de los permisos que queremos otorgar:

- **4**: Lectura (r)
- **2**: Escritura (w)
- **1**: Ejecución (x)

Sumando estos valores, especificamos los permisos que queremos otorgar. Por ejemplo:

- **7** (4+2+1): Lectura, escritura y ejecución (rwx).
- **6** (4+2): Lectura y escritura (rw-).
- **5** (4+1): Lectura y ejecución (r-x).
- **4**: Solo lectura (r--).

La sintaxis básica en formato numérico es:

```
chmod 755 archivo
```

En este ejemplo:

- El **7** otorga al propietario permisos de lectura, escritura y ejecución (rwx).
- El **5** otorga al grupo permisos de lectura y ejecución (r-x).
- El **5** otorga a los otros usuarios permisos de lectura y ejecución (r-x).

El formato simbólico es más legible y permite especificar los permisos de manera explícita, utilizando letras para indicar qué conjunto de usuarios se verá afectado y qué permisos se modificarán:

- **u**: Propietario (user).
- **g**: Grupo (group).
- **o**: Otros (others).
- **a**: Todos (all, es equivalente a ugo).

Los permisos se pueden añadir (+), eliminar (-) o establecer explícitamente (=).

La sintaxis básica en formato simbólico es:

```
chmod u=rwx,g=rx,o=rx archivo
```

En este ejemplo, estamos asignando:

- Al propietario (u), permisos de lectura, escritura y ejecución (rwx).

- Al grupo (g) y a otros (o), permisos de lectura y ejecución (r-x).

También podemos utilizar el formato simbólico para agregar o eliminar permisos sin afectar a los existentes:

```
chmod u+x archivo
```

Esto otorga al propietario (u) el permiso de ejecución (+x) sin modificar los otros permisos.

```
chmod g-w archivo
```

Esto elimina el permiso de escritura (-w) para el grupo (g), manteniendo intactos los demás permisos.

Además de cambiar los permisos de archivos y directorios, `chmod` ofrece opciones adicionales para aplicar los cambios de manera más eficiente. La opción `-R` (recursivo) permite aplicar los permisos de manera recursiva a todos los archivos y subdirectorios dentro de un directorio.

```
chmod -R 755 /ruta/del/directorio
```

Esto cambia los permisos del directorio y todo su contenido, lo que es útil cuando necesitamos cambiar los permisos de un conjunto grande de archivos.

Con la opción `--preserve-root` se asegura que no se cambien los permisos del directorio raíz accidentalmente, algo muy importante para proteger la integridad del sistema.

```
chmod -R --preserve-root 755 /
```

De esta manera, si intentamos cambiar los permisos recursivamente en el sistema de archivos completo, `chmod` evitará modificar los permisos del directorio raíz.

*Comando **chown***

El comando `chown` (change ownership) se utiliza en GNU/Linux para cambiar el propietario y/o el grupo de un archivo o directorio. Este comando es fundamental para la gestión de permisos y control de acceso, especialmente en entornos multiusuario, donde cada archivo o directorio tiene un propietario y un grupo asignado.

Cada archivo o directorio tiene dos propiedades principales:

1. **Propietario**: Es el usuario que posee el archivo o directorio y, por lo general, tiene el control completo sobre él.
2. **Grupo**: Es el grupo de usuarios que tiene permisos específicos sobre el archivo o directorio. Todos los usuarios que pertenecen a este grupo comparten los permisos definidos para el grupo.

La forma más simple de usar `chown` es cambiar el propietario de un archivo o directorio. La sintaxis es la siguiente:

```
chown usuario archivo
```

En este comando:

- **usuario**: Es el nombre del nuevo propietario.

- **archivo**: Es el archivo o directorio cuyo propietario queremos cambiar.

Podemos cambiar simultáneamente el propietario y el grupo de un archivo o directorio utilizando la siguiente sintaxis:

```
chown usuario:grupo archivo
```

Si deseamos cambiar solo el grupo, sin modificar el propietario, podemos omitir el nombre de usuario y dejar solo los dos puntos y el nombre del grupo:

```
chown :grupo archivo
```

El comando `chown` incluye varias opciones útiles que nos permiten aplicar los cambios de manera más flexible y eficiente. La opción `-R` (recursivo) permite cambiar el propietario y el grupo de manera recursiva en todos los archivos y subdirectorios dentro de un directorio.

```
chown -R juan:desarrolladores /ruta/del/directorio
```

Este comando cambia el propietario y el grupo de todos los archivos y directorios dentro de `/ruta/del/directorio`, incluidos los subdirectorios. Es útil para modificar permisos de manera masiva en estructuras de directorios complejas.

Con la opción `--reference` es posible cambiar el propietario y el grupo de un archivo o directorio para que coincida con los de otro archivo o directorio de referencia.

```
chown --reference=archivo_referencia archivo
```

En este caso, el archivo `archivo` adoptará los mismos permisos de propiedad y grupo que `archivo_referencia`. Esta opción es útil cuando queremos que varios archivos compartan la misma configuración de propiedad sin necesidad de especificar manualmente los detalles.

Cambiar el propietario de archivos y directorios es una operación delicada, especialmente en sistemas multiusuario o de servidores. Es importante verificar cuidadosamente los permisos y las configuraciones de seguridad para evitar errores que puedan exponer información o dar acceso no deseado a usuarios no autorizados.

Redirección de entrada y salida

En GNU/Linux, la línea de comandos trabaja con tres flujos básicos de datos que se utilizan para interactuar con el sistema y los programas. Estos flujos son la **entrada estándar** (stdin), la **salida estándar** (stdout), y el **error estándar** (stderr). La redirección nos permite **controlar cómo se manejan estos flujos** y es clave para automatizar tareas y manipular datos de manera eficiente.

Entrada, salida y error estándar

1. **Entrada estándar (stdin)**: La **entrada estándar** es el flujo de datos que un comando o programa **recibe**. Por defecto, la entrada estándar proviene del teclado, pero podemos redirigirla para que un programa reciba la entrada desde un archivo o incluso de otro comando. En la mayoría de los sistemas, la entrada estándar está identificada por el descriptor de archivo **0**.

2. **Salida estándar (stdout)**: La **salida estándar** es el flujo de datos que un programa o comando **envía como resultado de su ejecución**. De manera predeterminada, la salida estándar aparece en la terminal (pantalla), pero podemos redirigirla a archivos o a otros comandos para procesar los datos de manera más eficiente. El descriptor de archivo asociado con la salida estándar es el **1**.
3. **Error estándar (stderr)**: El **error estándar** es el flujo donde los programas o comandos **envían los mensajes de error**. Al igual que la salida estándar, los errores se muestran en la terminal por defecto, pero podemos redirigirlos para guardarlos en archivos separados o manejarlos de otra manera. El descriptor de archivo asociado con el error estándar es el **2**.

Redirección de salida

La redirección de la salida estándar nos permite controlar dónde queremos que aparezcan los resultados de un comando o programa. En lugar de mostrar la salida en la terminal, podemos guardarla en un archivo o enviarla a otro comando.

Con el operador >, podemos redirigir la salida de un comando a un archivo. Si el archivo ya existe, será **sobrescrito**.

```
comando > archivo.txt
```

Por ejemplo, si ejecutamos:

```
ls > listado.txt
```

El resultado del comando `ls` se guardará en el archivo `listado.txt` en lugar de mostrarse en la terminal. Si listado.txt ya existe, su contenido será reemplazado.

Si queremos **agregar** la salida de un comando a un archivo sin sobrescribir su contenido, utilizamos el operador >>. Por ejemplo, si ejecutamos:

```
echo "Nuevo contenido" >> archivo.txt
```

La línea `"Nuevo contenido"` se agregará al final de `archivo.txt`, conservando el contenido existente.

Redirección de entrada

La redirección de la **entrada estándar** nos permite enviar datos desde un archivo a un programa o comando en lugar de escribirlos manualmente en la terminal. Utilizamos el operador < para tomar la entrada de un archivo en lugar de ingresarla manualmente. Por ejemplo, si tenemos un archivo llamado `datos.txt` que contiene varias líneas de texto, podemos pasar ese archivo como entrada a un comando como `cat`:

```
cat < datos.txt
```

Esto mostrará el contenido de `datos.txt` en la terminal, como si lo estuviéramos escribiendo directamente.

Redirección de error

La redirección de la **salida de error estándar** nos permite manejar los mensajes de error por separado de la salida estándar. Esto es útil cuando queremos capturar los errores en

un archivo o suprimirlos sin afectar la salida principal del programa. Para redirigir solo los mensajes de error a un archivo, usamos el descriptor 2 seguido del operador >. Por ejemplo:

```
ls /directorio_no_existe 2> error.log
```

En este caso, si intentamos listar un directorio que no existe, el mensaje de error se guardará en el archivo `error.log` en lugar de mostrarse en la terminal. Podemos redirigir la salida estándar y los errores a archivos diferentes usando ambos descriptores (1 y 2):

```
comando > salida.txt 2> error.log
```

En este caso, los resultados exitosos del comando se guardarán en `salida.txt` y los mensajes de error se guardarán en `error.log`.

Si queremos redirigir tanto la salida estándar como los errores al mismo archivo, usamos el operador &>. Por ejemplo:

```
ls /directorio /directorio_no_existe &> salida_y_errores.txt
```

Aquí, tanto la lista de archivos de `/directorio` como el error por intentar acceder a `/directorio_no_existe` se guardarán en `salida_y_errores.txt`.

Redirigir hacia `/dev/null`

El archivo especial `/dev/null` se utiliza para descartar cualquier salida o error que redirijamos hacia él. Es conocido como el "agujero negro" del sistema, ya que cualquier cosa enviada allí desaparece. Por ejemplo:

```
comando > /dev/null
```

Esto ejecuta el comando, pero descarta toda la salida. Si queremos descartar los mensajes de error, entonces:

```
comando 2> /dev/null
```

Esto ejecuta el comando y descarta cualquier mensaje de error, permitiendo que la salida estándar continúe mostrándose normalmente.

Tuberías (Pipes)

En GNU/Linux, las **tuberías** (pipes) nos permiten conectar la **salida** de un comando directamente a la **entrada** de otro. De esta manera, podemos combinar varios comandos para realizar tareas complejas de forma eficiente y en una sola línea de comandos. Este concepto es fundamental para aprovechar al máximo la línea de comandos, ya que nos permite procesar datos de forma fluida sin necesidad de utilizar archivos temporales.

Concepto de tubería

La idea detrás de una tubería es que la **salida estándar** de un comando (lo que normalmente veríamos en la terminal) se redirige como **entrada estándar** para otro comando. Para crear una tubería, utilizamos el **operador de tubería** (|).

Por ejemplo, si usamos el comando `ls` para listar archivos y el comando grep para buscar un patrón dentro de esa lista, podemos encadenarlos con una tubería:

```
ls | grep archivo
```

En este caso:

1. `ls` genera una lista de archivos en el directorio actual.
2. La salida de `ls` se redirige como entrada a `grep`.
3. `grep` filtra esa lista, mostrando solo los nombres de archivos que contienen la palabra "archivo".

Uso del operador |

El operador de tubería (|) es el símbolo que usamos para conectar dos o más comandos en la terminal. Cuando colocamos el operador entre dos comandos, la salida del primer comando se convierte en la entrada del segundo.

La sintaxis básica es:

```
comando1 | comando2
```

Veamos un ejemplo sencillo:

```
cat archivo.txt | sort | uniq
```

En este caso:

1. **`cat archivo.txt`**: Muestra el contenido de archivo.txt.
2. **`sort`**: Ordena alfabéticamente el contenido del archivo.
3. **`uniq`**: Elimina las líneas duplicadas en la salida ordenada.

Todo esto se realiza en una sola línea, sin necesidad de crear archivos temporales.

Combinación de comandos con pipes: creación de flujos de trabajo eficientes

El verdadero poder de las tuberías se manifiesta cuando las usamos para combinar múltiples comandos en un flujo de trabajo eficiente. Al conectar varios comandos, podemos realizar operaciones complejas de manera concisa y directa.

Veamos algunos ejemplos más avanzados:

Contar el número de líneas que coinciden con un patrón:

```
grep "error" archivo.log | wc -l
```

Aquí estamos buscando la palabra "error" en `archivo.log` con `grep`. Luego, utilizamos `wc -l` para contar cuántas líneas contienen esa palabra. De esta forma, obtenemos rápidamente el número de errores en el archivo de registro.

Mostrar los 5 procesos más consumidores de CPU:

```
ps aux | sort -nrk 3,3 | head -n 5
```

En este caso, `ps aux` lista todos los procesos que se están ejecutando en el sistema. La salida de `ps aux` se envía a `sort -nrk 3,3`, que ordena los procesos por el uso de CPU (columna 3) en orden descendente. Finalmente, `head -n 5` muestra solo los 5 primeros resultados, que son los procesos que más CPU están utilizando.

Encontrar archivos grandes en un directorio:

```
du -h /ruta/del/directorio | sort -hr | head -n 10
```

En este ejemplo, `du -h` muestra el tamaño de los archivos y directorios en `/ruta/del/directorio`. Luego, `sort -hr` ordena estos archivos por tamaño en orden descendente, y `head -n 10` muestra los 10 archivos más grandes.

Casos prácticos: uso de pipes en la vida real

Las tuberías son ampliamente utilizadas en la administración de sistemas y en la automatización de tareas. Algunos ejemplos prácticos incluyen:

Monitoreo del sistema: Podemos combinar comandos como `top`, `ps`, `grep` y `awk` para crear flujos personalizados que nos permitan monitorear el uso de recursos del sistema o identificar procesos problemáticos.

```
ps aux | grep apache | awk '{print $2, $3, $4}'
```

En este caso, estamos buscando todos los procesos relacionados con apache y luego, con `awk`, extraemos las columnas 2 (PID), 3 (CPU) y 4 (memoria).

Extracción y manipulación de datos: En análisis de datos o grandes volúmenes de información, las tuberías nos permiten extraer, filtrar y transformar datos rápidamente.

Por ejemplo, para obtener una lista de direcciones IP únicas desde un archivo de registro:

```
cat access.log | awk '{print $1}' | sort | uniq
```

Aquí, `awk '{print $1}'` extrae la primera columna (las direcciones IP) de `access.log`, luego `sort` las ordena, y `uniq` elimina las duplicadas.

Automatización de backups: Al usar comandos como `tar`, `gzip`, y `scp`, podemos crear tuberías para realizar y transferir copias de seguridad automáticamente.

```
tar czf - /ruta/a/backup | ssh usuario@servidor "cat > /ruta/de/destino/backup.tar.gz"
```

En este ejemplo, estamos comprimiendo el contenido de `/ruta/a/backup` con `tar` y enviando la salida comprimida a través de `ssh` para guardarla en otro servidor. Todo esto sucede en una sola línea de comandos.

Resumen

Las **tuberías** en GNU/Linux son una herramienta poderosa para la **combinación de comandos** y la creación de flujos de trabajo eficientes. Utilizando el operador `|`, podemos encadenar comandos para realizar tareas complejas sin la necesidad de archivos intermedios. Desde el filtrado de datos hasta el monitoreo del sistema y la automatización de procesos, las tuberías son una técnica fundamental que nos ayuda a aprovechar al máximo la flexibilidad y el poder de la línea de comandos.

Expresiones Regulares (RegEx)

Las **expresiones regulares** (RegEx) son una herramienta extremadamente poderosa en GNU/Linux para buscar y manipular texto basado en patrones. Nos permiten realizar búsquedas avanzadas y reemplazos en archivos de texto o flujos de datos, combinando precisión y flexibilidad. Las expresiones regulares se utilizan en varios comandos, como `grep`, `sed` y `awk`, y nos ayudan a filtrar, modificar y extraer información de manera eficiente.

Introducción a las expresiones regulares: Sintaxis básica

Una expresión regular es una secuencia de caracteres que define un **patrón de búsqueda**. A través de este patrón, podemos buscar coincidencias en cadenas de texto. Por ejemplo, si quisiéramos encontrar todas las líneas que contienen la palabra "error" en un archivo de registro, usaríamos una expresión regular que busque esa palabra.

Un ejemplo básico usando `grep` sería:

```
grep "error" archivo.log
```

Esto buscará todas las líneas que contengan la palabra "error" en `archivo.log`.

Caracteres especiales y clases de caracteres

En las expresiones regulares, existen varios **caracteres especiales** que tienen un significado diferente al de los caracteres comunes. Estos nos permiten crear patrones más complejos y flexibles.

Punto (.)

Representa **cualquier carácter** excepto una nueva línea.

```
grep "a.b" archivo.txt
```

Esto coincidirá con cualquier cadena que tenga una "a", seguida de cualquier carácter, y luego una "b", como "acb", "a7b", o "a-b".

Caret (^)

Marca el **inicio de una línea**. Si queremos buscar líneas que comiencen con una palabra específica, usamos ^.

```
grep "^inicio" archivo.txt
```

Esto solo mostrará las líneas que comienzan con la palabra "inicio".

Signo pesos ($)

Marca el **final de una línea**.

```
grep "fin$" archivo.txt
```

Esto buscará líneas que terminen con la palabra "fin".

Asterisco (*)

Coincide con **cero o más ocurrencias** del carácter o patrón anterior.

```
grep "a*b" archivo.txt
```

Esto coincide con cualquier número de "a" seguido de una "b", por ejemplo, "b", "ab", "aaab", etc.

Signo más (+)

Coincide con **una o más ocurrencias** del carácter anterior.

```
grep "a+b" archivo.txt
```

Esto coincidirá con "ab", "aab", "aaab", etc., pero no con "b" solo.

Signo de interrogación (?)

Coincide con **cero o una ocurrencia** del carácter o patrón anterior.

```
grep "colou?r" archivo.txt
```

Esto coincidirá con "color" y "colour", haciendo opcional la "u".

Clases de caracteres

Las **clases de caracteres** nos permiten definir un conjunto de caracteres entre los cuales podemos hacer coincidir. Estas se encierran entre corchetes (`[]`).

Rango de caracteres: Podemos definir rangos utilizando guiones. Por ejemplo, [a-z] coincidirá con cualquier letra minúscula.

```
grep "[0-9]" archivo.txt
```

Esto buscará cualquier línea que contenga un número.

Clases predefinidas: Hay algunas clases de caracteres predefinidas que son útiles para patrones específicos:

- \d: Coincide con cualquier dígito (equivalente a [0-9]).
- \w: Coincide con cualquier carácter de palabra (letras, dígitos, y guiones bajos).
- \s: Coincide con cualquier espacio en blanco (espacios, tabulaciones, nuevas líneas).

Un ejemplo con `\d` para encontrar números:

```
grep "\d" archivo.txt
```

Repeticiones y patrones

Las **repeticiones** son una parte clave de las expresiones regulares que nos permiten controlar cuántas veces debe aparecer un carácter o grupo de caracteres para que coincida con el patrón. Estas se especifican utilizando llaves `{}`.

{n}: Coincide con exactamente **n ocurrencias** del carácter o patrón anterior.

```
grep "a{3}" archivo.txt
```

Esto buscará líneas que contengan exactamente tres "a" consecutivas, como "aaa".

{n,}: Coincide con **n o más ocurrencias** del carácter anterior.

```
grep "a{2,}" archivo.txt
```

Esto coincidirá con "aa", "aaa", "aaaa", etc.

{n,m}: Coincide con **entre n y m ocurrencias** del carácter o patrón anterior.

```
grep "a{2,4}" archivo.txt
```

Esto coincidirá con "aa", "aaa", o "aaaa", pero no con "a" ni con más de 4 "a".

Usos comunes en grep, sed y awk

Las expresiones regulares son extremadamente útiles cuando se combinan con comandos como `grep`, `sed` y `awk` para buscar y manipular texto.

grep: Se utiliza principalmente para **buscar** patrones en archivos. Ejemplo:

```
grep "^error" archivo.log
```

Esto mostrará todas las líneas en archivo.log que comiencen con la palabra "error".

sed: Es una herramienta de **edición de flujo** que nos permite buscar, reemplazar, insertar y borrar texto dentro de archivos o flujos de datos. Ejemplo de reemplazo:

```
sed 's/error/warning/' archivo.log
```

Esto reemplaza la primera aparición de la palabra "error" por "warning" en cada línea del archivo.

awk: Es una herramienta para **procesar y analizar** texto estructurado. Permite filtrar y manipular líneas basadas en expresiones regulares. Ejemplo:

```
awk '/error/ {print $1, $2}' archivo.log
```

Esto busca líneas que contengan la palabra "error" y muestra solo la primera y segunda columna.

Comodines (Globbing)

En GNU/Linux, los **comodines** (o globbing) son patrones que nos permiten **coincidir y seleccionar archivos** en función de un conjunto de caracteres o un patrón específico. A través de los comodines, podemos realizar operaciones como copiar, mover, listar o eliminar archivos sin necesidad de especificar cada nombre de archivo individualmente. Los comodines se utilizan comúnmente en la terminal y son parte del comportamiento nativo del shell.

Uso de los comodines en la terminal

Los comodines son símbolos especiales que se utilizan para **coincidir** con uno o más caracteres en los nombres de archivos o directorios. A continuación, revisaremos los más comunes:

Asterisco (*)

Coincide con cero o más caracteres. Es el comodín más flexible y utilizado.

```
ls *.txt
```

Este comando listará todos los archivos que terminen en .txt. El asterisco (*) coincide con cualquier nombre de archivo, sin importar su longitud, antes de la extensión.

Signo de interrogación (?)

Coincide con **exactamente un carácter**. Es útil cuando queremos buscar archivos que difieran en un solo carácter.

```
ls archivo?.txt
```

Este comando coincidirá con archivos como `archivo1.txt` o `archivoA.txt`, pero no con `archivo10.txt`, ya que el comodín ? solo coincide con un carácter.

Corchetes ([ ])

Coinciden con **uno de los caracteres** entre los corchetes. Podemos especificar una lista de caracteres o un rango.

Lista de caracteres:

```
ls archivo[abc].txt
```

Esto coincidirá con `archivoa.txt`, `archivob.txt`, y `archivoc.txt`, pero no con otros archivos.

Rango de caracteres:

```
ls archivo[0-9].txt
```

Esto coincidirá con `archivo1.txt`, `archivo2.txt`, etc., hasta `archivo9.txt`, porque 0-9 define un rango de dígitos.

Negación (!) dentro de corchetes

Coincide con cualquier carácter que **no** esté en la lista entre los corchetes.

```
ls archivo[!a-c].txt
```

Esto coincidirá con cualquier archivo que comience con "archivo" y termine en .txt, pero que no tenga las letras "a", "b", o "c" en esa posición.

Diferencias entre globbing y expresiones regulares

Aunque los **comodines** y las **expresiones regulares** parecen similares en algunos aspectos, son herramientas diferentes en cuanto a cómo se utilizan y qué capacidades ofrecen:

- **Comodines (Globbing)**:
    - Son interpretados por el shell antes de ejecutar un comando.
    - Están limitados a patrones de búsqueda simples.
    - Se usan comúnmente en operaciones con archivos (por ejemplo, `ls`, `cp`, `mv`, `rm`).
    - Ejemplos incluyen *, ?, y [].
- **Expresiones regulares**:

- Se utilizan en herramientas como `grep`, `sed`, y `awk`.
- Ofrecen patrones de búsqueda más complejos y poderosos (por ejemplo, coincidencia condicional, repetición).
- No son manejadas por el shell, sino por las herramientas que las implementan.
- Ejemplos incluyen `.` para cualquier carácter, ^ para el inicio de línea, y $ para el final de línea.

Ejemplos prácticos de globbing en la manipulación de archivos

Los comodines son muy útiles en la terminal para manipular varios archivos a la vez sin tener que especificar cada uno individualmente. Aquí algunos ejemplos prácticos:

- Listar todos los archivos de un tipo específico:

```
ls *.txt
```

  Este comando listará todos los archivos con la extensión .txt en el directorio actual.

- Copiar múltiples archivos con una estructura de nombres similar:

```
cp proyecto_202[0-9]* /backup/proyectos
```

  Esto copiará todos los archivos cuyo nombre comience con `proyecto_202` seguido de un dígito (como `proyecto_2020`, `proyecto_2021`, etc.) al directorio `/backup/proyectos`.

- Eliminar archivos que sigan un patrón:

```
rm temp_?.log
```

  Esto eliminará todos los archivos que comiencen con `temp_`, seguidos de un solo carácter (por ejemplo, `temp_1.log`, `temp_a.log`), pero no `temp_10.log`, ya que el comodín ? solo coincide con un carácter.

- Mover archivos dentro de un rango de nombres:

```
mv informe[1-3].pdf /home/informes_antiguos/
```

  Este comando moverá los archivos `informe1.pdf`, `informe2.pdf`, e `informe3.pdf` al directorio `/home/informes_antiguos/`.

- Filtrar archivos que no coincidan con un conjunto de caracteres:

```
ls proyecto[!abc].txt
```

  Esto listará todos los archivos que comiencen con proyecto, terminen en .txt, y que no tengan "a", "b", o "c" en esa posición específica.