
BASH Unidad 1

Shell, terminal y trabajar con comandos

Tabla de contenidos

¿Qué es la shell?

La shell ofrece lo que se denomina una interfaz de línea de comandos. Es un programa que **lee comandos introducidos por el usuario mediante el teclado y usa los servicios del sistema operativo para atender las peticiones**.

Por ejemplo, cuando abrimos a una shell para trabajar en ella, solemos ver algo así:

```
yo@mihost:~$
```

que se llama comúnmente *prompt* —en Microsoft Windows se usa el término símbolo del sistema en español o *command prompt* en inglés— y es la forma en la que el programa de la shell nos indica que está listo para recibir un comando.

El formato del *prompt* puede ser diferente en distintos sistemas y, por lo general, lo podemos ajustar a nuestro gusto. En el ejemplo anterior:

```
yo@mihost:~$
-- ------ -
|    |    |
|    |    +------- Directorio actual de trabajo. Las rutas
|    |             relativas lo serán respecto a esta ruta.
|    |
|    +------------ Nombre del equipo. Importante para no
|                  equivocarnos cuando estamos trabajando en
shells
|                  en distintos equipos al mismo tiempo.
|
+----------------- Nombre del usuario actual. Importante para no
                   equivocarnos, pues podemos trabajar en
distintas
                   shells usando cuentas de usuario diferentes.
```

Cuando la shell nos invita a darle una orden, solo tenemos que escribir un comando cualquiera y pulsar ENTER. Así sabe que hemos terminado de escribir el comando y que debe interpretar la orden y hacer lo que le hemos indicado. Por ejemplo:

```
yo@mihost:~$ cat hola.txt
```

que básicamente es un “abre el fichero `hola.txt` y muestra su contenido por pantalla”.

Ahora bien, la shell es un programa como cualquier otro. Para abrir y leer un fichero o para imprimir por la pantalla, necesita la intermediación del núcleo del sistema[1]. Los sistemas operativos Linux y compatibles POSIX ofrecen a los programas en C y C++ —entre otras opciones— la función open() para abrir y crear ficheros, read() para leer el contenido de los ficheros abierto y printf() para imprimir texto por pantalla. Así que la shell, al ver el comando anterior, podría:

1. Pedir al sistema operativo —mediante la función open()— que abra el fichero `hola.txt`. Una vez abierto...
2. Pedir al sistema operativo —mediante la función read()— que lea el contenido del fichero y lo copie en una zona de la memoria. Una vez leído...
3. Pedir al sistema operativo —mediante la función printf()— que imprima en la pantalla el contenido de la memoria con el contenido del archivo.

Al terminar, la shell vuelve a mostrar el *prompt* para indicar al usuario que está lista para aceptar un nuevo comando e interpretarlo.

La shell juega un papel importante para que un sistema sea útil. Por ejemplo, supongamos que no tenemos una shell y que, como antes, queremos ver el contenido del archivo `hola.txt`. Sin una shell en la que ejecutar `cat hola.txt`, no nos quedaría más remedio que echar mano de nuestros conocimientos de C y hacer un programa que ejecute los pasos comentados anteriormente. Ahora bien, eso no solo nos llevaría más tiempo, sino que presentaría un importante inconveniente. Para hacer nuestro programa necesitamos usar un editor de texto y luego ejecutar el compilador. ¿Cómo le vamos a pedir al sistema que nos abra el editor o compile nuestro programa si no hay una shell dedicada a esperar y atender nuestros órdenes?[2].

Hasta que comenzó a generalizarse el uso de interfaces gráficas de usuario (GUI) basadas en ventanas, las interfaces de línea de comandos (CLI) eran el tipo de interfaz más común. Hoy en día se siguen usando porque son más cómodas para ciertos tipos de tareas. Además, permiten crear de forma sencilla pequeños programas —llamados scripts— para automatizar trabajos rutinarios.

En la mayor parte de los sistemas Linux y POSIX actuales un programa llamado bash hace de shell, por lo que será esta con la que trabajaremos este curso. Sin embargo, no es ni mucho menos la única opción, ya que existen otras —por ejemplo ksh, tcsh o zsh— que pueden ser usados en su lugar según las preferencias de cada usuario. En macOS actuales la shell por defecto es zsh —aunque se puede cambiar, como ocurre en Linux—. Mientras que los sistemas Windows tienen dos opciones: el “Símbolo del sistema” o `cmd.exe`, que es la shell tradicional, y “Windows PowerShell”, que es una alternativa más moderna y recomendable.

---

[1] Esto suele ser así, excepto en sistemas realmente muy pequeños, donde puede implementarse algún tipo de interfaz de comandos muy sencilla en el núcleo del sistema operativo.
[2] Cuando se desarrolla para sistemas tan pequeños que no tienen interfaz, generalmente se usa un equipo convencional que si la tiene. Luego los archivos se copian al primero para hacer pruebas.

¿Qué es un terminal?

Tradicionalmente, se entiende por terminal o consola a la combinación de un teclado, para la entrada de comandos al ordenador, y una pantalla, para la exhibición de caracteres alfanuméricos —sin gráficos—. Originalmente, en la época de los primeros *mainframes* de tiempo compartido, estos terminales eran dispositivos independientes que se conectaban al *mainframe* por medio de una interfaz serie de comunicaciones.

Terminal IBM 3270 — Wikipedia

Emuladores de terminal

Hoy en día los ordenadores integran conexiones para teclado, ratón y otros dispositivos de entrada; lo que junto con las tarjetas gráficas actuales permite el uso de sistemas operativos con entornos gráficos de escritorio. Sin embargo, en dichos entornos es común encontrar un tipo de programa llamado emulador de terminal, que permite interactuar con una shell en el entorno gráfico. **Para que sea posible, los emuladores de terminal simulan el funcionamiento de un terminal no gráfico antiguo dentro de una ventana del entorno gráfico**.

Existen múltiples emuladores de terminal, algunos de los cuales acompañan a los diferentes entornos de escritorio disponibles en Linux: gnome-terminal, konsole, xterm, rxvt, kvt, nxterm, eterm, etc. Cuando abrimos cualquiera de estos emuladores de terminal, automáticamente se lanza una shell y podemos comenzar a interactuar con ella inmediatamente.

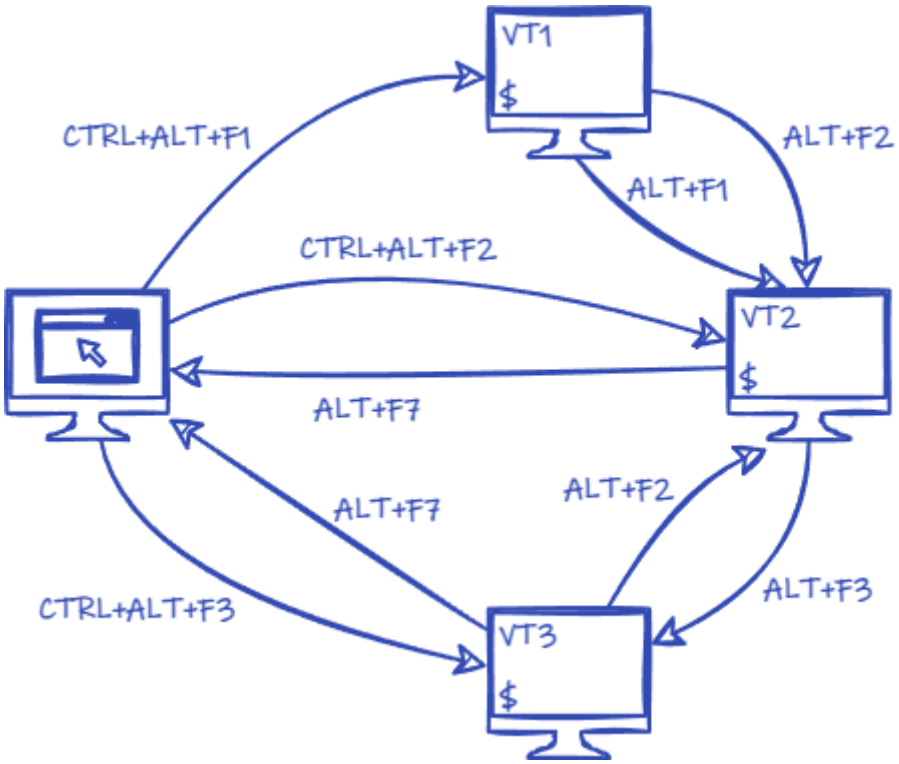
Otro ejemplo son las terminales integradas en los editores y entornos integrados de desarrollo (IDE) como: Visual Studio Code, Eclipse, IntelliJ IDEA / CLion o Visual Studio. Estos emuladores de terminal nos permite ejecutar comandos en una shell mientras estamos programando, sin abandonar la ventana de nuestro editor favorito.

Consola del sistema y terminales virtuales

En Linux, cuando nuestro sistema operativo no dispone de entorno gráfico —por ejemplo porque no lo hemos instalado— el arranque del sistema suele terminar mostrándonos la consola del sistema, que también emula un terminal no gráfico.

Ahora bien:

- Por lo general, esta consola del sistema en Linux no emula un único terminal, sino varias terminales virtuales, entre las que se puede conmutar usando la combinación de teclas ALT+F1, ALT+F2, etc.
- Si estamos en un entorno gráfico, podemos conmutar directamente a cualquiera de estas terminales virtuales incorporando CTRL a la combinación de teclas anterior. Es decir, pulsando: CTRL+ALT+F1, CTRL+ALT+F2, etc.
- Si nuestro sistema tiene entorno gráfico y queremos volver a él desde las terminales virtuales, el sistema suele estar configurado para que una de las terminales virtuales nos lleve de vuelta. Lo más común en los sistemas actuales es que sea la 7, por lo que simplemente tendremos que usar la combinación de teclas ALT+F7.



Comandos de la shell

Obviamente, cada shell admite unos comandos diferentes. En los sistemas operativos más pequeños el código de los comandos forma parte de la propia shell. Mientras que en los sistemas operativos más comunes —como Windows, Linux o macOS— unos pocos

---

comandos se pueden implementar en la propia shell —por los que se les denomina comandos *built-in*— pero la inmensa mayoría residen en sus propios ejecutables en el sistema de archivos. Eso significa que podemos extender la shell con nuevos comandos con solo crear nuevos programas.

Por ejemplo, al ejecutar:

```
yo@mihost:~$ cat hola.txt
```

la shell puede usar una implementación interna del comando cat. Si no la tiene —como ocurre en el caso de BASH— la shell buscará el ejecutable cat en el sistema de archivo y le pedirá al sistema operativo que cree un nuevo proceso y ejecute el programa del ejecutable en él.

Búsqueda de comandos

La shell no busca los comandos externos en todo el sistema de archivos, ya que eso podría consumir demasiado tiempo. En su lugar solo busca en los directorios indicados en la variable de entorno PATH —por el momento podemos pensar en ella como una variable de configuración de la shell—. Obviamente, esta variable se puede configurar manualmente para añadir otros directorios.

Para ver en BASH el contenido de la variable de entorno PATH podemos ejecutar lo siguiente:

```
yo@mihost:~$ echo $PATH
```

que puede mostrar una lista similar a esta:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games:/usr/local/games
```

En Linux y otros sistemas POSIX, la variable de entorno PATH contiene una lista de directorios, separados por ":" —en Windows es similar, pero se usa ";" como separador—. Para ejecutar un comando externo, la shell busca un ejecutable con el mismo nombre que el comando, recorriendo los directorios en el orden en que aparecen en la lista. En cuanto encuentra uno, lo ejecuta y deja de buscar.

Si queremos saber la ruta del ejecutable de un comando en concreto, solo tenemos que usar el comando which, que busca el ejecutable en los directorios de PATH y devuelve la ruta. Por ejemplo, para saber dónde está el ejecutable de cat:

```
yo@mihost:~$ which cat
/usr/bin/cat
```

Si queremos asegurarnos que la shell ejecuta un ejecutable concreto y no un comando built-in u otro ejecutable con el mismo nombre que aparezca antes en la lista de PATH, tenemos que indicar explícitamente la ruta del ejecutable que queremos ejecutar. Por ejemplo, si tuviéramos un cat mejorado en `/opt/bin` y quisiéramos asegurarnos que ejecutamos exactamente ese:

```
yo@mihost:~$ /opt/bin/cat hola.txt
```

Así, ya no cabe duda del programa que la shell ejecutará.

Argumentos de la línea de comandos

Cuando pulsamos ENTER, la shell lee la línea que hemos escrito y la divide por los espacios. La primera palabra es el nombre del comando que va a ejecutar, las siguientes son los argumentos del comando.

```
yo@mihost:~$ cat -n -A hola.txt mundo.txt README.me
             --- ---------------------------------
              |                    |
              |                    +----------------------
Argumentos
              |
              +------------------------------------------ Comando
```

Es importante tener en cuenta que la shell desconoce qué argumentos espera cada comando. No sabe si el comando espera nombres de archivos —como ocurre con cat — opciones —ni qué opciones admite— o simples palabras. Tampoco sabe si el orden de los argumentos es importante. Lo único que hace la shell es obtener las palabras de la línea de comandos y pedirle al sistema operativo que ejecute el ejecutable correspondiente con esos argumentos.

Si el programa del comando está implementado en C o C++, estos argumentos proporcionados por la shell son recibidos a través de la función main() del programa como cadenas de caracteres.

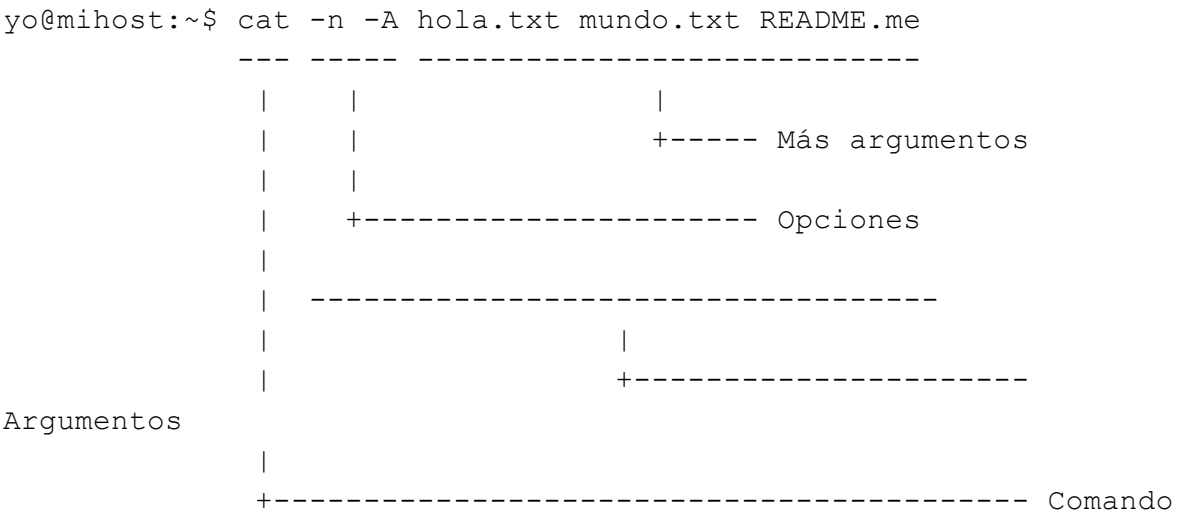
```
int main(int argc, char* argv[]) { ... }
             ----        ------
              |            |
              |            +------- Lista de argumentos
              |
              +------------------- Número de argumentos
```

Así que los programadores tienen total libertad a la hora de decidir cómo cada comando interpreta los argumentos y cómo avisa de los errores en caso de que el usuario no haya proporcionado argumentos correctos.

Sin embargo, en cada sistema podemos observar ciertos patrones comunes. Es decir, igual que en las aplicaciones de escritorio, esperamos que haya un menú **Archivo**, con opciones para abrir y guardar documentos y salir, o un menú **Editar** con los conocidos **Copiar**, **Cortar** y **Pegar**; los programadores de los comandos de la shell también suelen[3] respetar ciertas convenciones.

Veamos algunas:

- Los comandos suelen aceptar flags u opciones que modifican el funcionamiento del comando. En Linux, macOS y otros sistemas POSIX, las opciones suelen indicarse mediante un guion "-" seguido de una o varias letras. Mientras que en Windows se suele preferir el carácter "/".



- Muchos comandos usan opciones con una sola letra "-n". Se suele recomendar que si el comando acepta opciones que usan más de una letra, estas tengan que ir precedidas de un doble guion: "--number".

  Esto es muy útil porque muchos comandos aceptan que se compacten varias opciones de una letra en un solo argumento. Por ejemplo:

  ```
  yo@mihost:~$ cat -n -A hola.txt
  ```

  es equivalente a

  ```
  yo@mihost:~$ cat -nA hola.txt
  ```

  No hay confusión con opciones de más de una letra porque en ese caso espera dos guiones seguidos:

---

[3] Es importante entender que no hay ninguna obligación de respetar estas convenciones, por lo que resulta sencillo encontrar comandos que no lo hagan.

```
yo@mihost:~$ cat --number --show-all hola.txt
```

En todo caso, no todos los comandos respetan estas convenciones. Por ejemplo, el comando tar admite opciones con y sin guion. Mientras que el comando find soporta muchas opciones con nombres largos, pero precedidas por un único guion.

- En los nombres o letras de las opciones se suelen respetar ciertas convenciones. Por ejemplo, casi todos los comandos tienen una ayuda que se obtiene con la opción "-h" o "--help". Además, esta ayuda suele presentar en muchos casos un formato muy similar.

  Los comandos que tienen una opción para ejecutarse sin mostrar nada por pantalla suelen usar "-q" o "--quiet". Mientras que los comandos que admiten un modo donde aún dan más detalles de los usuales mientras se ejecutan, suelen usar "-v" o "--verbose" para activar esta posibilidad.

  Otro ejemplo es si el comando permite hacer algo de forma recursiva o al revés. Entonces suelen nombrar estas opciones como "-R" o "-r".

En muchos comandos el orden de los argumentos es irrelevante, pero no siempre ocurre así. Algunos comandos exigen que primero se indiquen las opciones y luego el resto de argumentos —como nombres de archivos u otros argumentos—. Además, algunas opciones esperan ir acompañadas de un argumento.

Por ejemplo, el comando tar se usa para comprimir. Admite una opción "-f" que se usa para indicar el nombre del archivo donde guardar los contenidos comprimidos:

```
yo@mihost:~$ tar -zc -f archivo_comprimido.tgz
/director/a/comprimir
```

por tanto, la opción "-f" siempre debe ir seguida del nombre de archivo.

Varios comandos en una línea

Por lo general se escribe un comando y se pulsa ENTER para ejecutarlo. Sin embargo, shells como bash permiten ejecutar varios comandos en una única línea. Solo hace falta que separemos los comandos mediante ";":

```
yo@mihost:~$ cd /etc; cat passwd; cd
```

La shell ejecuta los comandos uno detrás de otro, esperando a que el anterior termine antes de ejecutar el siguiente comando.

Historial de comandos

Los comandos ejecutados no se pierden, sino que se guardan en el historial, al que podemos acceder mediante el comando history:

```
yo@mihost:~$ history
 1988  cat hola.txt
 1989  echo $PATH
 1990  which cat
 1991  cat -n -A hola.txt mundo.txt README.me
 1992  cat -n -A hola.txt
 1993  cat -nA hola.txt
 1994  cat --number --show-all hola.txt
 1995  tar -zc -f archivo_comprimido.tgz /director/a/comprimir
 1996  cd /etc; cat passwd; cd
 1997  history
```

BASH soporta algunas combinaciones de teclas y comandos para recuperar rápidamente comandos en el historial. Si encontramos el comando que nos interesa, basta con pulsar ENTER para que se ejecute de nuevo:

Comando	**Acción**
Cursor ↑	Retroceder por los comandos del historial, desde comandos más recientes a más antiguos.
Cursor ↓	Avanzar por los comandos del historial.
CTRL+R *seguido de* <cadena>	Buscar, desde el comando más reciente hacia atrás, el primer comando que coincida con la cadena de búsqueda. Si el comando encontrado no es el que buscamos, se puede volver a pulsar CTRL+R para retroceder por el historial a un comando anterior que encaje.
!posición	Ejecuta el comando en la *posición* indicada del historial
!!	Vuelve ejecutar el último comando en el historial.
!cadena	Ejecuta el comando más reciente que empiece por la *cadena* de texto
!$ o ALT+.	Se sustituye por el último argumento del último comando en el historial. Por ejemplo, el comando **ls !$** ejecuta ls usando como primer argumento el último del comando

	anterior.
!*	Se sustituye por todos los argumentos del último comando en el historial. Por ejemplo, el comando **ls !*** ejecuta ls usando los mismos argumentos que se usaron en el comando anterior.

Trabajando con comandos

Como comentamos anteriormente, los comandos pueden ser de distintos tipos:

- **Un programa ejecutable** como aquellos que vimos en /usr/bin. En esta categoría entran tanto binarios compilados, escritos en lenguajes como C o C++, como programas escritos en lenguajes de script como los de la propia shell, Perl, Python, Ruby, etc.
- **Un comando interno incluido en la propia shell (built-in)**. BASH proporciona una serie de comandos internos, de los cuales por el momento solo hemos visto el comando cd.
- **Una función de la shell**. Son mini scripts de shell que se definen para permanecer cargados en el entorno. Los veremos más adelante en el curso.
- **Un alias**. Comandos que se definen a partir de otros comandos. También los veremos más adelante en el curso.

Identificar comandos

A veces es útil saber exactamente a cuál de los cuatro tipos anteriores pertenece un comando.

type

El comando interno **type** muestra qué clase de comando es el comando indicado: alias, función, comando interno, palabra clave o programa ejecutable.

```
yo@mihost:~$ type cd
cd is una orden interna de la shell

yo@mihost:~$ type ls
ls es un alias de 'ls --color=auto'

yo@mihost:~$ type cp
cp is /bin/cp
```

El comando **type** básico está incluido en el estándar POSIX, pero muchas shells que siguen este estándar —incluida BASH— adicionalmente aceptan varias opciones que puede ser muy útiles, aunque no sean parte del estándar:

- **type -t comando**: Muestra solo el tipo de comando usando una única palabra: *alias*, *builtin*, *file*, *keyword* o *function*.
- **type -p comando**: Si el comando es un programa ejecutable, muestra solo la ruta al archivo ejecutable. Si el comando es de otro tipo, no muestra nada.
- **type -P comando**: Como "-p" pero forzando la búsqueda del comando en los directorios de la variable PATH; ignorando si hay un alias, función o comando interno con el mismo nombre[4][5].
- **type -a comando**: Muestra todas las definiciones posibles del comando. Por ejemplo, si un comando está definido como comando interno y como programa ejecutable —como suele ser el caso del comando **echo**— se mostrará la información de ambas definiciones.

Por tanto, si queremos determinar la localización en el sistema de archivos de un programa ejecutable como **ls**, podemos usar **type** así:

```
yo@mihost:~$ type -P ls
/usr/bin/ls
```

Solución de problemas comunes

Durante el trabajo con la shell es común encontrarse con ciertos errores o situaciones problemáticas. En esta sección veremos los problemas más frecuentes y sus soluciones.

"Command not found" / "Orden no encontrada"

Este es uno de los errores más comunes para usuarios principiantes. Cuando la shell muestra este mensaje, significa que no puede encontrar el comando que hemos solicitado.

```
yo@mihost:~$ lst
bash: lst: command not found
```

Posibles causas y soluciones:

- **Error al deletrear el comando**: Es la causa más frecuente. En el ejemplo anterior, probablemente queríamos escribir "ls" en lugar de "lst".
- **El programa no está instalado**: Si estamos seguros de que hemos escrito correctamente el comando, es posible que el programa no esté instalado en el sistema. Podemos verificarlo usando el comando **type**:

  ```
  yo@mihost:~$ type -P lst
  ```

  Si no devuelve ninguna ruta, puede que el programa no esté instalado o que esté instalado, pero en un directorio no incluido en la variable PATH, con el resultado de

---

[4] Algunos sistemas traen una utilidad llamada which que tiene un funcionamiento similar al de **type -P**, solo que no se trata de un comando estándar.
[5] La forma recomendada por el estándar POSIX para conocer la ruta de un comando es utilizar **command -v comando**.

---

que la shell no es capaz de encontrar el ejecutable del comando.

- **El programa no está en el PATH**: Aunque el programa esté instalado, puede que no se encuentre en ninguno de los directorios listados en la variable PATH que, como explicamos anteriormente, se utiliza para configurar los directorios donde la shell busca programas ejecutables.

  Podemos verificar el contenido de PATH con:

  ```
  yo@mihost:~$ echo $PATH
  ```

  Si conocemos la ubicación del programa, siempre podemos ejecutarlo indicando la ruta completa. Por ejemplo:

  ```
  yo@mihost:~$ /opt/mi_proyecto/bin/mi_comando
  ```

"Permission denied" / "Permiso denegado"

Este error aparece cuando intentamos ejecutar un comando, pero no tenemos los permisos necesarios.

```
yo@mihost:~$ mi_comando
bash: mi_comando: Permission denied
```

Posibles causas y soluciones:

- **El archivo no tiene permisos de ejecución**: Podemos verificar primero dónde está ubicado el comando y luego comprobar sus permisos:

  ```
  yo@mihost:~$ type -P mi_comando
  /usr/local/bin/mi_comando
  yo@mihost:~$ ls -l /usr/local/bin/mi_comando
  -rw-r--r-- 1 root root 2458 sep 5 10:30 mi_comando
  ```

  Lo veremos con más detalle en la Unidad 2, pero si no aparece el permiso "x" (permiso de ejecución) para nuestro usuario, es que no tenemos permisos para ejecutar el programa.

- **No tenemos permisos para acceder al directorio**: Si intentamos ejecutar un programa en un directorio al que no tenemos acceso, recibiremos este error.
- **Necesitamos privilegios de administrador**: Algunos comandos requieren permisos de superusuario. En este caso, como también veremos en la Unidad 2, podemos usar los comandos **su** o **sudo**.

  Por ejemplo, para ejecutar el comando **reboot** con permisos de superusuario:

```
yo@mihost:~$ sudo reboot
```

Comandos que "se cuelgan"

A veces un comando parece no responder o tarda mucho más de lo esperado. En estos casos tenemos varias opciones:

- **CTRL+C**: Interrumpe la ejecución del comando actual. Es la opción más común para detener un programa que no responde.

```
yo@mihost:~$ comando_muy_lento
^C
yo@mihost:~$
```

- **CTRL+Z**: Suspende el comando actual, enviándolo al segundo plano. El comando queda pausado y podemos retomarlo más tarde:

```
yo@mihost:~$ comando_muy_lento
^Z
[1]+ Stopped comando_muy_lento
```

- **CTRL+D**: Envía una señal de fin de archivo (EOF). Útil cuando un programa está esperando leer de la entrada del usuario, para indicarle que ya no hay nada más que leer, por lo que el programa seguramente terminará:

```
yo@mihost:~$ cat
^D
yo@mihost:~$
```

Pantalla desordenada

Si la pantalla se ve desordenada o muestra caracteres extraños:

- **CTRL+L**: Limpia la pantalla del terminal.
- **reset**: Reinicia completamente el terminal a su estado inicial:

```
yo@mihost:~$ reset
```

Obteniendo documentación acerca de los comandos

Ahora veremos cómo obtener documentación y ayuda acerca de cada uno de los comandos disponibles.

--help

Prácticamente, todos los comandos soportan la opción "--help" que muestra una descripción de la sintaxis del comando y sus opciones.

```
yo@mihost:~$ mkdir --help
Modo de empleo: mkdir [OPCIÓN]... DIRECTORIO...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short
options too.
  -m, --mode=MODE   establece los permisos (como en chmod), en
                    lugar de  a=rwx - umask
  -p, --parents     no hay error si existen, crea los directorios
                    padres en caso necesario
  -v, --verbose     muestra un mensaje por cada directorio creado
  -Z, --context=CTX  establece el contexto de seguridad SELinux de
                       cada directorio creado a CTX
      --help     muestra esta ayuda y finaliza
      --version  informa de la versión y finaliza
```

Hay algunas cosas que debemos tener en cuenta acerca de la ayuda:

- Cuando en la descripción de un comando aparecen elementos entre corchetes, se nos está indicando que tales elementos son opcionales.
- Si una barra vertical separa a varios elementos, se nos está indicando que dichos elementos son exclusivos. Es decir, que si usamos uno no podemos usar el otro.

Por lo tanto, si la ayuda dice:

```
cd [-L|-P] [dir]
```

lo que nos indica es que el comando cd puede ir seguido opcionalmente de la opción "-L" o "-P" pero no de ambas. Y que el argumento "dir" también es opcional.

Si conocemos el comando que queremos utilizar y solo necesitamos refrescar cómo se invoca o alguna opción en particular, "--help" puede resultarnos muy útil. Si, por el contrario, queremos aprender a utilizar el comando, seguramente sea mejor consultar su *página del manual* o cualquier otra documentación de ayuda que tenga.

man

Muchos programas ejecutables diseñados para ser usados desde la línea de comandos proporcionan un tipo de documentación denominado *página del manual*. Se puede acceder a la página del manual de un programa concreto usando el comando:

```
yo@mihost:~$ man ls
```

Las páginas del manual por lo general no incluyen ejemplos, por lo que están más pensadas para ser usadas como una referencia que como un tutorial para aprender a usar un programa.

Navegar por las páginas

En la mayor parte de los sistemas Linux man usa el programa less para mostrar la documentación, por lo que, para navegar por las páginas del manual, se pueden usar los siguientes comandos y teclas rápidas de less:

Comando	**Acción**
Re. Pág. o b	Retroceder una página.
Av. Pág. o espacio	Avanzar una página.
G	Ir al final del archivo.
1G	Ir al principio del archivo.
/caracteres	Buscar la siguiente ocurrencia de los *caracteres* especificados.
n	Repetir la búsqueda hecha previamente.
h	Mostrar la lista completa de comandos y opciones soportadas por el programa.
q	Salir

Secciones del manual

El manual se divide en secciones. Las lista de secciones se puede obtener en la *página de manual* del manual:

```
yo@mihost:~$ man man
```

O con el siguiente comando

```
yo@mihost:~$ man -f intro
intro (8)    - introduction to administration and privileged
commands
intro (3)    - introduction to library functions
intro (1)    - introduction to user commands
intro (7)    - introduction to overview and miscellany section
intro (2)    - introduction to system calls
intro (5)    - introduction to file formats and filesystems
intro (6)    - introduction to games
intro (4)    - introduction to special files
```

Al buscar la página de un programa, man recorre las secciones y se detiene en la primera página que encuentra. Esto puede ser un problema porque a veces existen términos similares en secciones distintas. Si sabemos en qué sección está la página que nos interesa, podemos indicarlo en la línea de comandos de man.

Por ejemplo, si sabemos que estamos buscando un comando llamado **write**, podemos indicar explícitamente que solo mire en la sección 1, que es la de los comandos:

```
yo@mihost:~$ man 1 write
```

Si nos interesa una llamada al sistema llamada **write**, podemos indicar la sección 2:

```
yo@mihost:~$ man 2 write
```

Y si se trata de una función de C, podemos pedirle que mire en la sección 3:

```
yo@mihost:~$ man 3 printf
```

Si queremos que mire en todas las secciones y nos muestra todas las páginas, usamos la opción "-a".

```
yo@mihost:~$ man -a write
```

De esta forma abrirá un less con la primera página que encuentre y al cerrarla, si hay otra página con el mismo nombre en otra sección, nos volverá a abrir less con ella.

Además de usar el comando man, **una forma muy cómoda y recomendable de consultar el manual es a través de los sitios web https://man7.org/linux/man-pages o http://linux.die.net.**

Manual de BASH

Al igual que muchos otros programas, BASH tiene una extensa página de manual, donde se explican tanto sus opciones de línea de comandos como la sintaxis del lenguaje de script o sus comandos internos:

```
yo@mihost:~$ man bash
```

Es importante tener en cuenta que **los comandos internos de BASH no tienen una página de manual propia**. Por ejemplo, si pedimos la página del comando cd:

```
yo@mihost:~$ man cd
Ninguna entrada del manual para cd
```

La ayuda de comandos internos de BASH, como "cd", "help" y "read", entre muchos otros, está en la página de manual de BASH. Por tanto, si nos interesa, tenemos que abrir la página de BASH y buscar en ella la documentación del comando correspondiente.

help

BASH dispone de ayuda integrada para cada uno de sus comandos internos. Para acceder a ella solo es necesario escribir "help" seguido del comando en cuestión:

```
yo@mihost:~$ help cd
cd: cd [-L|[-P [-e]]] [dir]
    Modifica el directorio de trabajo del shell.

    Modifica el directorio actual a DIR.  DIR por defecto es el
    valor de la variable de shell HOME.

    ...
```

Si lo preferimos, también podemos incluir la opción "-m" para obtener la ayuda en un formato más similar al del comando man.

```
yo@mihost:~$ help -m cd

NAME
    cd - Modifica el directorio de trabajo del shell.

SYNOPSIS
    cd [-L|[-P [-e]]] [dir]

DESCRIPTION
    Modifica el directorio de trabajo del shell.

    ...
```

apropos

El comando man no resulta muy práctico cuando no conocemos el nombre exacto del comando o programa del que queremos obtener ayuda. En ese caso, se puede usar el comando apropos para que busque en las descripciones de todas las *páginas del manual* el término que le indiquemos, devolviendo un listado con los nombres y secciones de las páginas donde haya una coincidencia:

```
yo@mihost:~$ apropos zip
bunzip2 (1)          - a block-sorting file compressor, v1.0.8
bzcmp (1)            - compare bzip2 compressed files
bzdiff (1)           - compare bzip2 compressed files
bzegrep (1)          - search possibly bzip2 compressed files
for...
bzfgrep (1)          - search possibly bzip2 compressed files
for...
bzgrep (1)           - search possibly bzip2 compressed files
for...
bzip2 (1)            - a block-sorting file compressor, v1.0.8
bzip2recover (1)     - recovers data from damaged bzip2 files
bzless (1)           - file perusal filter for crt viewing of...
bzmore (1)           - file perusal filter for crt viewing of...
```

```
docker-context-import (1) - Import a context from a tar or zip
file
funzip (1)              - filter for extracting from a ZIP archive...
gpg-zip (1)             - encrypt or sign files into an archive
gunzip (1)              - compress or expand files
gzip (1)                - compress or expand files
Regexp::Common::zip (3pm) - - provide regexes for postal codes.
streamzip (1)           - create a zip file from stdin
unzip (1)               - list, test and extract compressed files
in...
unzipsfx (1)            - self-extracting stub for prepending to
ZIP...
zforce (1)              - force a '.gz' extension on all gzip files
zip (1)                 - package and compress (archive) files
zipcloak (1)            - encrypt entries in a zipfile
zipdetails (1)          - display the internal structure of zip files
zipgrep (1)             - search files in a ZIP archive for lines...
zipinfo (1)             - list detailed information about a ZIP
archive
zipnote (1)             - write the comments in zipfile to stdout,...
zipsplit (1)            - split a zipfile into smaller zipfiles
```