CHULETA SHELL DE LINUX

CONCEPTOS, COMANDOS, SINTAXIS Y SCRIPTS

VISTA POR SECCIONES

¿QUÉ ES UN SHELL?

ELEMENTOS DEL SHELL

FICHEROS

FICHEROS REGULARES

DIRECTORIOS

OTROS FICHEROS

PERMISOS

ENTRADA/SALIDA

EXPRESIONES REGULARES Y EDICIÓN DE FLUJO

PROCESOS

GESTIÓN DE USUARIOS Y GRUPOS

ADMINISTRACIÓN DE DISCOS

ADMINISTRACIÓN DE PAQUETES EN DEBIAN

COMANDOS DE RED

OTROS COMANDOS

EXPANSIONES

VARIABLES Y ALIAS

SENTENCIAS DE CONTROL Y BUCLES

SCRIPTS Y FUNCIONES SHELL

VISTA POR COMANDOS

- .
- [
- addgroup
- adduser
- alias
- apt
- basename
- bc
- bg
- case
- cat
- cd
- chage
- chgrp
- chmod
- chown
- clear
- cp
- crontab
- curl
- cut
- date
- delgroup
- deluser
- df
- diff
- dirname
- dpkg
- du
- echo
- env
- eval
- exec
- exit
- export
- fdisk
- fg
- file
- find
- for
- getopts
- grep
- head
- help
- history
- id
- if
- info
- ip
- jobs
- jq
- kill
- less
- ln
- local
- logger
- login
- logout
- ls
- lsblk
- man
- mkdir
- mkfs
- mktemp
- more
- mount
- mv
- nano
- nc
- netcat
- nice
- nohup
- nslookup
- passwd
- ping
- poweroff
- printenv
- ps
- pwd
- read
- readlink
- realpath
- reboot
- return
- rm
- rmdir
- scp
- sed
- seq
- set
- sftp
- shift
- shutdown
- sleep
- snap
- sort
- source
- ss
- ssh
- stat
- su
- sudo
- tail
- tar
- tee
- test
- top
- touch
- tr
- traceroute
- type
- umask
- umount
- unset
- wait
- wget
- which
- while
- who
- xargs

VISTA POR OPERADORES Y EXPANSIONES

!	(	>	{ (agrup.)	${
!!	((	>&	{ (expans.)	
!*n*	*	>>	|	
"	:	?	||	
'	;	[	~	
#	<	[[	$	
&	<<	\	$(	
&&	=	`	$((	

DEFINICIÓN, INTÉRPRETES SHELL WINDOWS Y LINUX

¿QUÉ ES UN SHELL?

¿QUÉ ES UN SHELL?

Un **shell** es una interfaz con el sistema operativo.

Gestiona la interacción entre el usuario y el sistema operativo solicitándole una entrada (comandos), interpretándola y gestionando el resultado de salida.

Normalmente cuando hablamos de shell nos referimos a la línea de comandos del sistema, pero la interfaz gráfica de usuario también entraría en la definición de un shell.

En este documento nos centraremos en el shell de línea de comandos o **terminal Linux**.

INTÉRPRETES SHELL WINDOWS

Los intérpretes de línea de comandos que usa Windows son:

- **Símbolo del sistema (CMD):** es el intérprete de línea de comandos de Windows desde Windows NT.
- **PowerShell:** ofrece características avanzadas además de las del símbolo del sistema, como ejecución remota, automatización de tareas y mucho más. Ha sido liberado como código abierto y ofrece versiones para Linux y Mac.

INTÉRPRETES SHELL UNIX

Los intérpretes shell clásicos más conocidos para sistemas operativos tipo UNIX son:

- **Shell Bourne (sh o bsh):** shell compatible con el estándar POSIX (Portable Operating System Interface) creado por Steve Bourne, es el shell predeterminado para el sistema operativo Unix.
- **Shell C (csh):** diseñado para permitir a los usuarios escribir programas en script de shell con una sintaxis parecida al lenguaje C. Es incompatible con POSIX. Tiene una versión mejorada, TC (**tcsh**), el shell por defecto en los sistemas operativos basados en BSD, como FreeBSD.
- **Shell Korn (ksh):** combina las características de csh y bsh. Empezó como software propietario.
- **Shell Almquist (ash):** shell ligera compatible con bsh pero sin muchos de los extras que aportan otras alternativas.

INTÉRPRETES SHELL UNIX

Actualmente, los shells más extendidos en sistemas tipo UNIX son:

- **Shell Bourne-Again (bash):** versión de código abierto GNU actualizada del shell Bourne original que incorpora algunas de las funcionalidades de csh y ksh, y otras propias. Es el shell interactivo usado por defecto en casi todas las distribuciones Linux, como Debian, Arch Linux, etc. Se puede encontrar en **`/bin/bash`**.
- **Shell Debian Almquist (dash):** implementación del shell Bourne (sh) más limitada que bash pero mucho más rápida (4x) y ligera. Es descendiente de ash. No sirve de shell interactiva, sino que se usa para ejecutar scripts shell más eficientes, aunque no todas las construcciones bash son compatibles con dash. `dash` reemplaza a `sh`, por lo que a menudo ejecuta los scripts que empiezan por `#!/bin/sh`.

INTÉRPRETES SHELL UNIX

Otros shells:

- **Shell Z (zsh):** poderoso shell interactivo compatible con POSIX con características de bash, ksh y tcsh. Es una de las alternativas a bash más populares que hay actualmente. Es el shell por defecto de Kali Linux y macOS X.
- **Shell Friendly Interactive (fish):** incluye características como autosugestión, resaltado de sintaxis o historial de búsqueda. Es incompatible con POSIX.
- **BusyBox**: shell interactivo derivado de ash que contiene solo lo necesario para cualquier sistema pequeño o embebido. Es el shell de la distribución Alpine Linux.
- Otros como **elvish**, **yash**, **ion**, **osh**, **nash**, etc.

INTÉRPRETES SHELL UNIX

Los intérpretes shell instalados en un sistema tipo UNIX los podemos encontrar en el directorio `/bin`.

> Todos estos sistemas tienen en común la existencia de `/bin/sh`, pero no es el `sh` original, sino que suele ser un enlace simbólico a otro intérprete, como `bash`, `ash`, `dash`, `busybox`, etc.

Para ver los intérpretes shell disponibles (al menos los que terminan en `sh`):

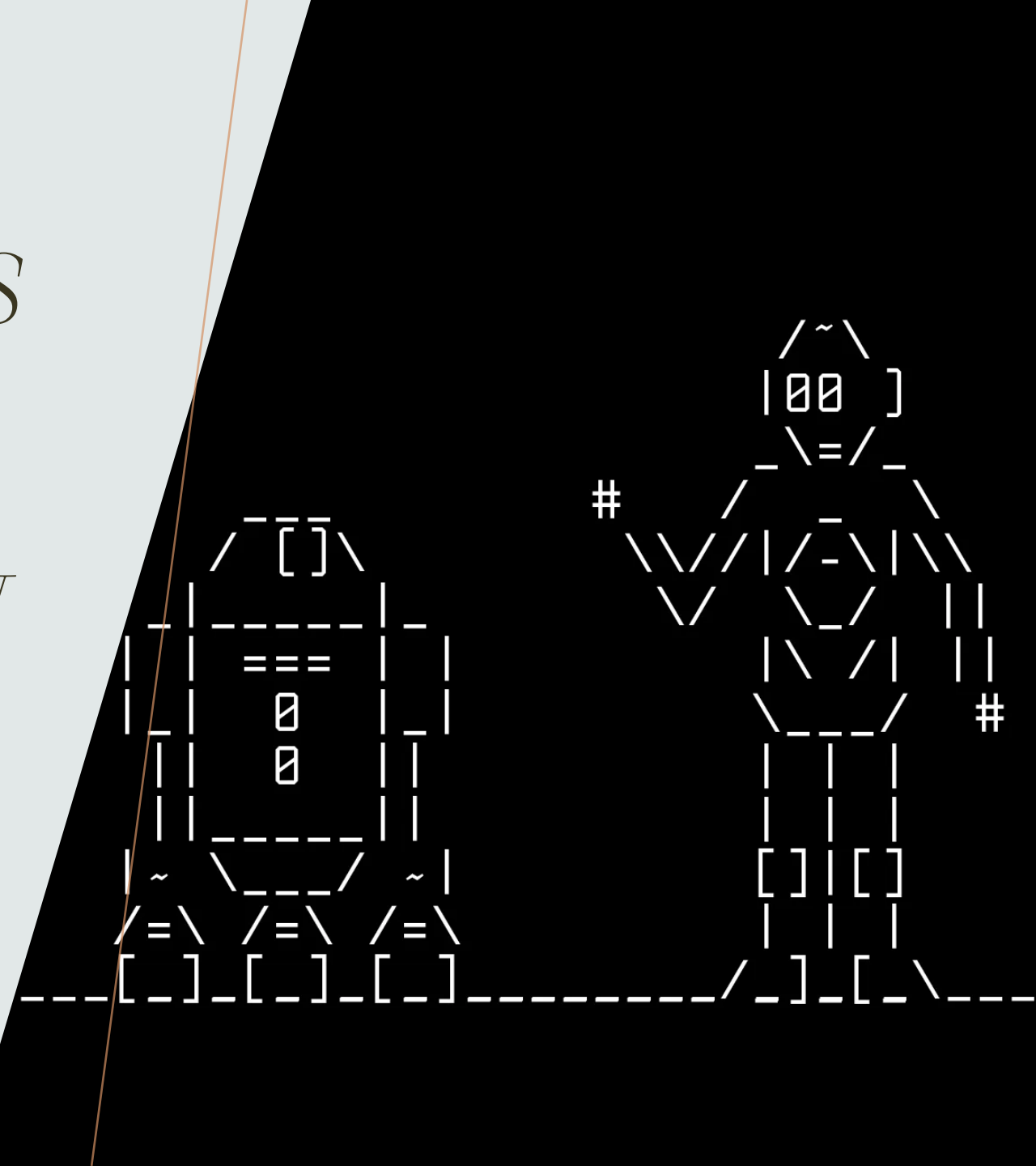
```
$ ls -l /bin/*sh
```

Puedes ver el intérprete del shell actual mediante la variable `SHELL`:

```
$ echo $SHELL
```

EN ESTE DOCUMENTO NOS CENTRAREMOS EN LA SINTAXIS COMPATIBLE CON ***POSIX***

DIFERENCIAREMOS ENTRE LA SINTAXIS COMPATIBLE CON POSIX Y LA ESPECÍFICA DE GNU BASH



PROMPT, HISTORIAL, FORMATO DE UN COMANDO, AGRUPAMIENTO, COMENTARIOS, CARACTERES DE ESCAPE, COMILLAS, AYUDA, ATAJOS, EXPRESIONES ARITMÉTICAS

COMANDOS HISTORY, MAN

ELEMENTOS DEL SHELL

PROMPT

En la siguiente salida (sacada de Ubuntu):

```
user@host:~$ _
```

`user@host:~$` es el **prompt** del shell. Muchos sistemas operativos tienen un prompt característico.

El prompt se puede modificar mediante ciertas variables (`PS0-PS4` según nivel, `PS1` para principal).

Es costumbre personalizar el prompt en el archivo `.bashrc` (o equivalente).

Otros ejemplos de prompts son:

```
[root@host ~]# _
```

```
> _
```

En general, los prompts de este documento se representarán así:

```
$ _
```

Para representar un prompt con permisos de administrador, se usará:

```
# _
```

HISTORIAL

Los intérpretes interactivos suelen guardar un historial de comandos ejecutados.

En bash, puedes acceder al comando anterior pulsando `Arriba`, y al comando siguiente (si lo hay) pulsando `Abajo`.

Existe un comando para revisar los comandos que se introdujeron en la sesión shell:

```
$ history
```

También podemos repetir el último comando con **`!!`**, repetir el comando número `n` con **`!n`** y repetir el enésimo comando anterior con **`!-n`**.

FORMATO DEL COMANDO SHELL

Un **comando** en el shell tiene la forma:

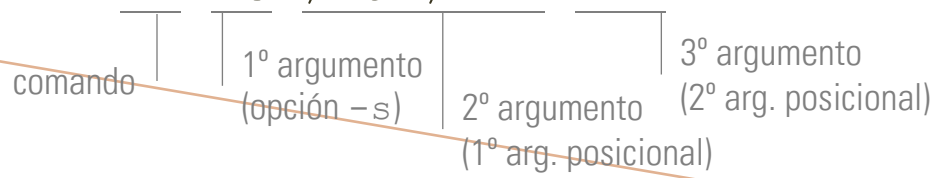
```
$ <ejecutable> [<argumentos...>]
```

Es decir, el nombre del comando (o archivo ejecutable) y, opcionalmente, una lista de argumentos separados por espacios.

Generalmente existen dos tipos de argumentos:

- **Opciones**: suelen empezar por un guión (–) o dos (––). Se pueden escribir separados (`-d -c`) o juntos (`-dc`). Pueden ir solos (`-o`) o tener su propio argumento posicional (`-o 5`, `-o=5`, `-o5`).
- **Argumentos posicionales**: se suelen poner después de las opciones, y tienen un orden determinado: cambiar su orden suele cambiar el significado del comando (`cp f1 f2` ≠ `cp f2 f1`).

Ejemplo:



AGRUPAMIENTO DE COMANDOS

Los comandos se pueden agrupar mediante:

```
(<comandos...>)
```

o mediante:

```
{ <comandos...>; }
```

La primera forma ejecuta los comandos en un nuevo subshell idéntico al shell actual, mientras que la segunda los ejecuta en el shell actual (un poco más eficiente).

Agrupar comandos permite redireccionar su salida como si se tratase de un solo comando. Ejemplo:

```
$ { printf "hello " ; printf " world\n" ; } > saludo
```

SEPARADORES DE COMANDOS

Dos comandos del shell se pueden separar mediante:

- **Salto de línea** (`\n`): una línea por cada comando.
- **Punto y coma** (`;`): permite escribir varios comandos en una sola línea.
- **Tubería** (`|`): conecta la salida estándar izquierda con la entrada estándar derecha.
- **Ampersand** (`&`): lanza el proceso de la izquierda en segundo plano.
- **Operadores lógicos AND** (`&&`) u **OR** (`||`): ejecuta o no el comando de la derecha según el éxito o fracaso del de la izquierda.

El comando especial `:` no realiza ninguna acción.

COMENTARIOS

Un **comentario**, en cualquier lenguaje de programación, es un texto que no va a ser ejecutado. En el shell, los comentarios empiezan por # y terminan donde termina la línea. Ejemplo:

```
$ cp fil fil1 # Comentario al final del comando
$ # El siguiente comando muestra todos los procesos activos:
$ ps -e
```

Es recomendable poner comentarios descriptivos en los scripts para que cualquiera pueda entender lo que hacen al leerlos.

Shebang es un tipo de comentario especial que se pone en la primera línea de un script.

CARACTERES DE ESCAPE

En el shell existen **caracteres que tienen un significado especial** (espacios, saltos de línea, >, <, |, &, $, *, etc.), pero a veces necesitamos usarlos como **caracteres literales**.

Para decirle al intérprete shell que trate a un carácter especial como texto literal (es decir, que lo ignore), podemos **escapar** dicho carácter anteponiéndole una barra invertida (\) de la siguiente forma: \c, donde c es el carácter especial.

Por ejemplo:

```
$ echo 7 > 5  # redirección
$ echo 7 \> 5 # > es ignorado
7 > 5
```

También se puede ignorar un salto de línea escribiendo \ al final de la línea:

```
$ echo El comando continúa \
en la siguiente \
línea
El comando continúa en la
siguiente línea
```

CARACTERES ENTRECOMILLADOS

Si lo que queremos es que el shell ignore una secuencia de caracteres, es preferible el uso de comillas. Existen dos tipos:

- **Comillas simples** (`' '`): el shell ignora todos los caracteres que estén encerrados entre comillas simples. Por ejemplo:

```
$ echo 'HOLA \\    $SHELL'
HOLA \\    $SHELL
```

- **Comillas dobles** (`" "`): el shell ignora todos los caracteres encerrados entre comillas dobles excepto `$` (permite expansiones), `` ` ``, `\` y `!`. Por ejemplo:

```
$ echo "HOLA \\    $SHELL"
HOLA \    /bin/bash
```

AYUDA

Para obtener ayuda con un comando, su uso, argumentos y salidas:

- Para **built-ins** y **palabras clave** del shell: `help`. Por ejemplo:

  ```
  $ help [[
  ```

- Para **archivos ejecutables** del sistema: `man` o `info`. Por ejemplo:

  ```
  $ man cat
  ```

Para saber de qué tipo es un comando se usa el built-in `type`, que nos dice si un comando es un ejecutable, alias, un built-in, una palabra clave, etc.:

```
$ type <comando>
```

Para conocer la ruta absoluta de un ejecutable:

```
$ which <comando>
```

MAN

Abre el **manual de uso** de un comando, función o archivo de configuración:

```
$ man curl
```

También se puede especificar en qué sección del manual buscar (en caso de que el nombre coincida con un manual de otra sección):

```
$ man 5 passwd
```

Se pueden buscar con manuales mediante expresiones regulares con `-k`:

```
$ man -k json
```

Principales secciones del manual:

- `1`: Comandos del shell.
- `2`: Llamadas al sistema (funciones de C).
- `3`: Llamadas a librerías (funciones de C).
- `4`: Archivos especiales (en `/dev`).
- `5`: Archivos de configuración (en `/etc`).
- `7`: Miscelánea.
- `8`: Comandos del shell de administración.

ATAJOS DEL TECLADO CONOCIDOS

- `Arriba/Abajo`: permite recorrer el historial de comandos recientes.
- `Tab`: autocompleta si puede (1x) o muestra las opciones posibles (2x).
- `Ctrl + Shift + C` / `Ctrl + Shift + V`: copiar/pegar (solo en terminales GUI).
- `Ctrl + C`: envía una señal para interrumpir (terminar) el proceso en primer plano.
- `Ctrl + Z`: envía una señal para detener el proceso en primer plano.
- `Ctrl + D`: Inserta el carácter End Of File (`EOF`), útil para cerrar una entrada estándar o cerrar una sesión.
- `Ctrl + L`: Limpia la pantalla. Equivale al comando `clear`.

OPERACIONES ARITMÉTICAS

Se pueden realizar operaciones aritméticas en el shell mediante el operador `(( ))`:

```
$ (( <expr> ))
```

Donde `<expr>` puede ser cualquier combinación de sumas (+), restas (–), productos (*), divisiones (/) o potencias (**, solo bash).

Ejemplos:

```
$ n=5 x=$(( 1+3*n ))
$ echo $x          #16
$ echo $((x**3)) #4096
```

El comando `bc` realiza funciones de calculadora algo más avanzadas.

TIPOS DE FICHEROS, RUTAS

COMANDOS FILE, CP, MV, RM, STAT, DU, TAR, MKTEMP

FICHEROS

TIPOS DE FICHERO

En los sistemas UNIX existen los siguientes tipos de fichero:

- **Ficheros regulares (`-`):** son los archivos que contienen programas, texto o datos binarios.
- **Directorios (`d`):** contienen otros ficheros y la información relativa a ellos.
- **Enlaces simbólicos (`l`):** apuntan a la ruta de un fichero. Se crean con `ln -s`.
- Ficheros especiales (suelen estar en `/dev`)
  - **Ficheros de dispositivos por bloques (`b`):** comunican con el hardware y los periféricos principalmente.
  - **Ficheros de dispositivos por caracteres (`c`):** representan un flujo de datos de entrada o salida, como la terminal, puertos serie, `/dev/random`, etc.
  - **Tuberías con nombre o FIFO (`p`):** sirven para enviar caracteres de un proceso a otro.
  - **Sockets (`s`):** se usan para intercambiar información entre aplicaciones.

FILE

Muestra el tipo de archivo:

```
$ file <fichero>
```

CP

Copia archivos a un destino (si es un directorio, dentro del directorio):

```
$ cp <origen> <destino>
$ cp <orígenes...> <directorio destino>
```

La opción `-r` es necesaria si se copian carpetas, para que se copie también su contenido.

La opción `-f` fuerza la copia (en caso de exista el archivo de destino y no sea editable).

La opción `-s` crea links simbólicos en lugar de hacer copias. La opción `-l` crea links físicos.

La opción `-u` (de update) no copia un archivo si ya existe y es más reciente que el del origen.

Ejemplos:

```
$ cp -r file1 file2 dir1/ dir2/
$ cp file1 file1.copia
```

MV

Mueve o renombra un archivo:

```
$ mv <origen> <destino>
$ mv <orígenes...> <directorio destino>
```

Ejemplos:

```
$ mv file.txt file.txt.backup
$ mv index.html style.css /var/www/html/
```

RM

Borra ficheros:

```
$ rm [-r] <rutas...>
```

Con la opción `-r` (de recursivo) permite borrar directorios y su contenido.

Con la opción `-f` intenta forzar el borrado aun cuando los archivos no tienen permiso de escritura.

Ejemplos:

```
$ rm file1 file2
$ rm -r /var/www/*
```

STAT

Muestra los metadatos de uno o varios ficheros:

```
$ stat <ficheros...>
```

Ejemplo:

```
user@host:~$ stat /usr/bin
  File: /usr/bin
  Size: 36864           Blocks: 72         IO Block: 4096   directory
Device: 252,0   Inode: 1179651     Links: 2
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2024-10-21 13:09:09.486716814 +0200
Modify: 2024-10-19 19:33:27.320756614 +0200
Change: 2024-10-19 19:33:27.320756614 +0200
 Birth: 2024-09-17 17:56:21.384931778 +0200
```

DU

Muestra el tamaño de un fichero en disco. Si es un directorio, muestra el tamaño de todos los directorios contenidos en él recursivamente:

```
$ du <opciones> <ruta...>
```

La opción `-s` da solo el tamaño total de cada uno de los ficheros especificados.

La opción `-c` añade un tamaño total, la suma de todos los ficheros especificados.

La opción `-h` muestra el tamaño en formato legible por un usuario.

La opción `-b` muestra el tamaño en bytes.

Ejemplo:

```
$ du -sh /home
$ du -cb /usr/bin/*sh
```

TAR

Este comando permite crear y trabajar con archivos tar, con o sin compresión.

Para empaquetar (`-c`) archivos en un nuevo fichero tar:

```
$ tar -cvf <fichero_tar> <archivos...>
```

Si al crear un archivo empaquetado se desea usar un algoritmo de compresión, se añade una de las siguientes opciones:

- `-z` o `--gzip` para **gzip** (`.tar.gz`): ofrece una compresión aceptable.
- `-J` o `--xz` para **xz** (`.tar.xz`): más lento en comprimir, pero mayor compresión.
- `--zstd` para **zstd** (`.tar.zst`): buena relación rapidez-compresión.
- `-a` para decidir el algoritmo de compresión según la extensión del fichero tar.

TAR

Opciones:

- `-f <fichero_tar>` especifica el fichero empaquetado o comprimido que se va a crear, leer o modificar.
- `-v` (verbose) para imprimir la lista de archivos que procesa.
- `-g <fichero>` crea una nueva copia incremental de los archivos, especificando un fichero de snapshot donde se especifican los cambios de los archivos. Si no existe, lo crea.
- `-C <ruta>` cambia de directorio antes de realizar las siguientes operaciones. El orden de esta opción con respecto al resto de opciones y argumentos importa.

TAR

Para **extraer** (`-x`) en la ruta actual o en una ruta especificada:

```
$ tar -xvf <fichero_tar> [-C <ruta>]
```

Se pueden especificar en `-x` los ficheros que se desea extraer:

```
$ tar -vf <fichero_tar> -x <ficheros...>
```

Para **ver el contenido** (`-t`) de un fichero tar (`-v` para ver más detalles):

```
$ tar -tf <fichero_tar> [-v]
```

Para **ver diferencias** (`-d`) entre el contenido de un fichero tar y un directorio (el directorio actual o el especificado con `-C`):

```
$ tar -df <fichero_tar> [-C <ruta>]
```

TAR

Los siguientes comandos solo son válidos para ficheros tar sin comprimir:

- Para **añadir archivos o reemplazarlos si existen** (`-r`) a un fichero tar:

```
$ tar -rvf <fichero_tar> <ficheros...>
```

- Para **añadir archivos** o **añadir contenido a archivos existentes** (`-u`, de update) de un fichero tar:

```
$ tar -uvf <fichero_tar> <ficheros...>
```

- Para **eliminar archivos** dentro del fichero tar:

```
$ tar -vf <fichero_tar> --delete <ficheros...>
```

MKTEMP

Crea un fichero temporal vacío en `/tmp` e imprime la ruta del fichero por la salida estándar:

```
$ mktemp
```

Es una opción útil y recomendable para los scripts que necesiten poner datos temporales en algún fichero.

Con la opción `-d`, **crea un directorio temporal** en lugar de un fichero regular.

Fallará si no puede crear el fichero.

El uso típico es guardar el nombre del fichero temporal en una variable, usar el fichero mediante la variable y luego borrarlo:

```
$ TMPFILE="$(mktemp)"
$ echo data > $TMPFILE
$ #...
$ rm $TMPFILE
```

COMANDOS TOUCH, NANO, CAT, LESS, MORE, TAIL, HEAD

FICHEROS REGULARES

TOUCH

Establece las fechas *access*, *modify* y *change* de uno o varios ficheros, o los crea (vacíos) si no existen:

```
$ touch <ruta...>
```

Ejemplos:

```
$ touch f1.txt f2.txt
$ touch index.html
```

NANO

Editor de ficheros sencillo:

```
$ nano <fichero>
$ nano +<lin>[,<col>] <fichero>
```

Si se especifica línea y/o columna, lo abre por esa posición.

Otros editores alternativos son `pico`, `vi`, `vim`, `emacs`, etc.

Ejemplos:

```
$ nano newfile
$ nano -lcm /etc/hosts
```

Algunas opciones son:

- `-l`: muestra los números de línea a la izquierda.
- `-c`: muestra línea y columna actual.
- `-m`: permite interacción con el ratón.
- `-B`: crea un backup (añadiéndole un ~ al nombre) al guardarlo.

Una vez abierto:

- Para guardar, `Ctrl+O` (da a elegir el nombre de destino y hay que pulsar `Enter`)
- Para salir, `Ctrl+X`.

CAT

Con**cat**ena y muestra el contenido de archivos por la salida estándar:

```
$ cat <archivos...>
```

Se suele usar para volcar el contenido de un solo archivo:

```
$ cat file1
```

Si se usa sin argumentos o con un guión (–), lee de la entrada estándar:

```
$ cat
```

Con la opción `-n`, se incluyen los números de línea, y con la opción `-e` se muestran los caracteres no imprimibles.

LESS

Abre un fichero para leerlo, o, si no se le pasa el fichero, la entrada estándar:

```
$ less [-N] [<fichero>]
```

La opción `-N` muestra números de línea a la izquierda.

Una vez abierto:

- Para moverte por el archivo, pulsa `Arriba` o `Abajo`.
- Para ir al principio/fin del archivo, pulsa la tecla `Inicio/Fin`
- Para salir, pulsa `q`.

Ejemplo:

```
$ less /var/log/syslog
$ set | less #podemos usarlo para visualizar textos muy largos
```

MORE

Abre un fichero para leerlo, pero paginado:

```
$ more <fichero>
```

Una vez abierto:

- Para moverte por las páginas, pulsa `Arriba` o `Abajo`.
- Para salir, pulsa `q`.

Ejemplo:

```
$ more /var/log/syslog
```

TAIL

Muestra el final de uno o más ficheros (sin argumentos lee de la entrada estándar):

```
$ tail [<archivos...>]
```

La opción `-n <num>` muestra las últimas `<num>` líneas del archivo (si `<num>` es negativo, imprime todas menos las `<num>` primeras).

La opción `-c <num>` muestra los últimos `<num>` bytes del archivo (si `<num>` es negativo, imprime todos menos los `<num>` primeros).

La opción `-f` actualiza el contenido automáticamente si el archivo sufre cambios.

Ejemplo:

```
$ tail -f /var/log/syslog
```

HEAD

Muestra el principio de uno o más ficheros (sin argumentos lee de la entrada estándar):

```
$ head <archivo...>
```

La opción `-n <num>` muestra las primeras `<num>` líneas del archivo (si `<num>` es negativo, imprime todas menos las `<num>` últimas).

La opción `-c <num>` muestra los primeros `<num>` bytes del archivo (si `<num>` es negativo, imprime todos menos los `<num>` últimos).

Ejemplo:

```
$ head /var/www/html/index.html
$ head -c 64 </dev/urandom
```

ÁRBOL DE DIRECTORIOS, RUTAS, ARCHIVOS OCULTOS

COMANDOS DIRNAME, BASENAME, LS, CD, PWD, MKDIR, RMDIR, FIND

DIRECTORIOS

ÁRBOL DE DIRECTORIOS RAÍZ

En sistemas tipo UNIX, existe un único **directorio raíz**, **`/`**, por lo que todas las rutas están contenidas en este directorio (empiezan por `/`). Esto es lo que llamamos **árbol de directorios raíz**.

Cada vez que se monta un sistema de archivos al árbol raíz, se incrusta la raíz de su árbol de directorios en un directorio del árbol de directorios raíz.

> Por ejemplo, `/mnt/usbdrive`, `/media/user/kingston64gb`, etc.

El directorio raíz tiene carpetas del sistema como `/etc` (configuración global, scripts de inicio, etc.), `/var` (archivos variables del sistema), `/tmp` (archivos temporales), `/bin` (ejecutables), `/dev` (dispositivos), `/home` (carpetas personales de usuarios), etc.

RUTAS

Una **ruta**, en sistemas UNIX, es una secuencia de directorios contenidos unos dentro de otros y separados por /, que sirve para acceder a un fichero.

Una ruta puede ser absoluta o relativa.

Una **ruta absoluta** empieza por `/` (directorio raíz) y llega hasta el fichero o directorio al que se refiere. Una ruta absoluta se refiere al mismo fichero independientemente de cuál sea el directorio actual (variable PWD). Ejemplo:

```
/etc/init.d/cron
```

Esta ruta se refiere a un archivo de nombre `cron` dentro del directorio `/etc/init.d`, que está contenido en `/etc`, que está contenido en `/` (raíz).

RUTAS RELATIVAS

Una **ruta relativa** no empieza por `/` y no puede localizar ficheros por sí misma, sino que se completa (según el contexto) para formar una ruta absoluta.

- La mayoría de los comandos son rutas relativas a archivos ejecutables que se buscan en las rutas de la variable `PATH`. La ruta absoluta de un comando se vería con:

  ```
  $ which <comando>
  ```

  En el caso de `ls` en Ubuntu, `which` lo localizará en `/usr/bin/ls`.

- Cuando escribimos rutas relativas en argumentos, redirecciones a archivos, etc., se asume que son relativas al directorio actual de trabajo (variable `PWD`). Ejemplo:

  ```
  $ cat dir/f1 >f2
  ```

  Si estamos en `/root`, por ejemplo, `dir/f1` y `f2` son rutas relativas a `/root`, por lo que `dir/f1` se refiere a `/root/dir/f1` y `f2` se refiere a `/root/f2`.

DIRNAME/BASENAME

Para extraer el **directorio padre** de una ruta (no es necesario que exista un fichero en dicha ruta):

```
$ dirname <ruta>
```

Para extraer el **nombre** de un fichero o directorio de una ruta:

```
$ basename <ruta>
```

Por ejemplo:

```
$ dirname /etc/network/interfaces
/etc/network
$ basename /etc/network/interfaces
interfaces
```

ARCHIVOS OCULTOS

Cuando el nombre de un archivo empieza por `.` (punto), es un archivo **oculto**.

Algunos comandos los ignoran si no se les incluye una opción especial (`ls -a`).

Existen dos archivos ocultos predefinidos en todos los directorios:

- `.` (punto): se refiere al directorio que lo contiene.
- `..` (punto punto): se refiere al directorio padre.

Ejemplo:

```
user@host:~$ cd /var/www/html
user@host:/var/www/html$ cd ../..
user@host:/var$ ./script.sh
```

LS

Lista el contenido del/los directorios seleccionados o del directorio actual:

```
$ ls [<ruta...>]
```

Ejemplos:

```
$ ls -l /etc/default
$ ls -lah ~
$ ls -ld .
$ ls
```

Opciones:

- `-l`: muestra la lista en formato largo:

  Incluye tipo de archivo, permisos, nº de enlaces, usuario y grupo propietarios, tamaño y fecha de última modificación.

- `-a`: incluye archivos ocultos (los que empiezan por `.`).
- `-h`: muestra los tamaños en kB, MB, etc.
- `-d`: si se pide listar un directorio, muestra este, no su contenido.

CD

El built-in `cd` (*Change Directory*) cambia el directorio actual de trabajo:

```
$ cd [<ruta>]
```

Ejemplos:

```
$ cd /var/log #ruta absoluta
$ cd dir1     #ruta relativa
```

Para ir al directorio personal (`$HOME`):

```
$ cd
```

Para ir al directorio padre:

```
$ cd ..
```

Para ir al directorio anterior (es decir, el directorio previo al último `cd`):

```
$ cd -
```

El shell solo recuerda un único directorio anterior (variable `OLDPWD`).

PWD

El built-in `pwd` (*Print Working Directory*) muestra la ruta del directorio actual de trabajo:

```
$ pwd
```

Es equivalente a imprimir el valor de la variable PWD:

```
$ echo $PWD
```

MKDIR

Crea un directorio vacío:

```
$ mkdir [-p] <ruta...>
```

Con la opción `-p`, si se especifica una ruta con directorios intermedios inexistentes, se crean en lugar de dar error, y si el directorio final existe, no hace nada (es decir, se asegura de que, al finalizar el comando, esa ruta entera exista).

Ejemplos:

```
# mkdir /srv/www /srv/www/asir.com
# mkdir -p /var/log/app/proc
```

RMDIR

Borra un directorio vacío:

```
$ rmdir <ruta...>
```

Para borrar un directorio no vacío, se usa `rm -r`.

FIND

Potente comando para encontrar archivos en un árbol de directorios:

```
$ find [-L] <rutas...> [<criterios...>] [<acciones...>]
```

En `<rutas...>` se pone el directorio (o directorios) donde se iniciará la búsqueda.

En `<criterios...>` se pueden especificar los criterios de búsqueda.

En `<acciones...>` se pueden especificar acciones (`-delete`, `-exec`) a ejecutar para cada elemento encontrado por `find`.

Ejemplos típicos:

```
$ find . -name '*.txt'
$ find /etc -maxdepth 1 -type l
```

FIND (CRITERIOS)

Principales **criterios de búsqueda**:

- `-name/-iname <patrón>`: nombre de fichero coincide con patrón.
- `-regex/-iregex <regexp>`: la ruta coincide con expresión regular.
- `-path/-ipath <patrón>`: la ruta de fichero coincide con patrón.
- `-lname/-ilname <patrón>`: es un enlace simbólico y la ruta a la que apunta coincide con patrón.

`i`: *case insensitive*, ignora mayúsculas (`F=f`).

- `-type [d|f|l|...]`: tipo del archivo (directorio, regular, enlace simbólico, ...).
- `-size [+|-]<n>[c|k|M|G]`: tamaño mayor (+), menor (–) o igual que `<n>` (en Bytes, kB, MB, GB).
- `-empty`: el fichero o directorio es vacío.
- `-uid/-gid <n>`: UID/GID propietario del fichero.
- `-user/-group <nombre>`: nombre de usuario/grupo propietario del fichero.

FIND (CRITERIOS)

- `-`*`x`*`min/-`*`x`*`time [+|-]<n>`: fecha de último *x* es de hace más (+), menos (–) o exactamente `<n>` minutos/días.
- `-[a|c]newer <fichero>`: es más reciente (a|c|m) que `<fichero>`.
- `-newer`*`x`*`t <fecha>`: fecha de último *x* es posterior a `<fecha>`.
- `-samefile <fich>`: comparten inodo.

x: a -> access, c -> change, m -> modify

- `-links [+|-]<n>`: el número de enlaces duros es mayor (+), menor (–) o igual que `<n>`.
- `-readable/-writable/-executable`: es legible/escribible/ejecutable por el usuario actual.
- `-perm [-]<máscara>`: tiene al menos (–) o exactamente los permisos definidos en `<máscara>`, que puede ser octal (`640`) o simbólica (`u=rw,g=r`).

FIND (ACCIONES Y OTROS)

Principales **acciones** (se ejecutan por cada archivo encontrado):

- `-print`: imprime la ruta terminada en un salto de línea (`\n`) (acción por defecto).
- `-print0`: imprime la ruta terminada en un carácter nulo (`\0`).
- `-delete`: borra el archivo o directorio.
- `-exec <comando...> ';'`: ejecuta un comando con sus argumentos, sustituyendo cada `{}` por el nombre del archivo. Debe terminar en `';'`.

Opciones generales:

- `-L`: fuerza a seguir enlaces simbólicos.
- `-mindepth/-maxdepth <n>`: mínima/máxima profundidad en que se buscará o se ejecutarán acciones.

  La profundidad `0` son los directorios pasados como argumentos (`<rutas...>`).

  `-mindepth` y `-maxdepth` se colocan después de las rutas y antes de los criterios de búsqueda.

FIND (EXPRESIONES)

Los criterios de búsqueda (y las acciones) se pueden combinar mediante los siguientes **operadores lógicos**:

- `\( <expr> \)`: precedencia (los paréntesis deben escaparse o entrecomillarse)
- `! <expr>`: negación
- `<expr> [-a] <expr>`: AND lógico (`-a` es opcional)
- `<expr> -o <expr>`: OR lógico

Ejemplos:

```
$ find /lib /usr/lib -name 'lib*.so' -a -type f
$ find /var -size +5G -o -mmin -20 -exec cp {} /mnt/usb/ ';'
```

ENLACES FÍSICOS Y SIMBÓLICOS, ARCHIVOS ESPECIALES EN /DEV

COMANDOS LN, READLINK, REALPATH

OTROS FICHEROS

ENLACES FÍSICOS

En Linux, lo que entendemos como rutas a ficheros en realidad no son más que **enlaces físicos** (o enlaces duros) que apuntan a la verdadera localización del archivo (en el sistema de archivos) mediante un **inodo**.

Normalmente, cada inodo tiene exactamente un enlace físico apuntándole. En este caso, al borrar el enlace físico, se elimina el archivo. Si un archivo tiene varios enlaces físicos, al borrar uno de ellos, el archivo permanece.

Un enlace físico solo puede apuntar a un inodo **de su propia partición**.

No se pueden crear enlaces físicos de directorios.

ENLACES SIMBÓLICOS

Un **enlace simbólico** (o enlace blando) es un tipo de fichero que apunta a una **ruta**, no a un inodo, por lo que no influye en la cuenta de enlaces a un archivo y puede apuntar a rutas de particiones diferentes a la suya. Borrar enlaces simbólicos no tiene repercusiones sobre el archivo al que apuntan

Si se elimina el archivo al que apunta un enlace simbólico, la ruta a la que apunta no cambia, por lo que fallará al ser accedida.

Un enlace simbólico es el equivalente UNIX al *acceso directo* de Windows.

LN

Crea un **enlace físico** a un archivo o directorio:

```
$ ln <objetivo> <nombre link>
```

Con la opción `-s`, crea un **enlace simbólico** (recomendado).

Ejemplo:

```
$ ln -s ../sites-available/asir.conf /etc/apache2/sites-enabled/
```

READLINK, REALPATH

Para extraer el destino de un enlace simbólico:

```
$ readlink <symlink>
```

Para resolver una ruta entera en otra sustituyendo los enlaces simbólicos por directorios/ficheros finales a los que apuntan:

```
$ realpath <ruta>
```

La ruta que se imprime es la **ruta real** al fichero, sin enlaces simbólicos.

Se permite que el último elemento de la ruta no exista.

ARCHIVOS ESPECIALES (/DEV)

Existen **archivos por caracteres** (`c`) especiales en `/dev` para ciertos fines, como:

- **`/dev/null`**: descarta los caracteres que se escriben en él. Si se lee de él, está vacío.

```
# adduser andy staff >/dev/null
```

- **`/dev/zero`**: cuando se lee de él, devuelve un flujo sin fin de **caracteres nulos (bytes con valor 0)**. Se puede usar para crear un flujo con cualquier carácter:

```
$ tr '\0' 'a' </dev/zero | head -c 32; echo
```

- **`/dev/random`**: cuando se lee de cualquiera de ellos, devuelve un flujo de **bytes pseudo-aleatorios criptográficamente seguros**, recolectando entropía de distintas fuentes. Por ejemplo, para generar una contraseña aleatoria de 32 caracteres imprimibles:

```
$ tr -dc '[:print:]' </dev/random | head -c 32; echo
```

TIPOS DE PERMISOS

COMANDOS SUDO, SU, CHMOD, CHOWN, CHGRP Y UMASK

PERMISOS

TIPOS DE PERMISOS

En los sistemas tipo UNIX, los permisos básicos de cualquier archivo se clasifican en tres tipos, que tienen un uso u otro dependiendo del tipo de archivo:

- **Lectura (`r`, `4`):** permite leer el contenido del fichero / ver el contenido de la carpeta.
- **Escritura (`w`, `2`):** permite modificar el contenido del fichero / añadir, editar (metadatos) o eliminar archivos de una carpeta.
- **Ejecución (`x`, `1`):** permite ejecutar el fichero / atravesar la carpeta.

Estos permisos se aplican a tres clases de usuarios diferentes:

- **Usuario (`u`):** asigna permisos al usuario propietario del archivo.
- **Grupo (`g`):** asigna permisos a cualquier usuario del grupo propietario del archivo.
- **Otros (`o`):** asigna permisos a cualquier otro.

PERMISOS POR BITS

Los permisos se pueden representar como bits (o en octal) de la siguiente manera:

```
rwxrwxrwx = (4+2+1, 4+2+1, 4+2+1) = 777
```

Donde los tres primeros bits corresponden a los permisos de usuario (`u`), los tres siguientes a los de grupo (`g`) y los tres siguientes a los de los demás (`o`).

Cuando un bit está a `1`, el permiso que representa está activo y se representa con su letra. Cuando está a `0`, el permiso no está activo y se representa con – (guión). Ejemplo:

```
rw-r--r-- = (4+2+0, 4+0+0, 4+0+0) = 644
```

En este archivo, el usuario propietario puede leer y escribir, todos los demás solo pueden leer y nadie puede ejecutar.

PERMISOS ESPECIALES

Set UID (SUID, `4000`): cuando un fichero tiene activo este bit (`u+s`), quien lo ejecute tendrá los mismos permisos que el usuario que creó este archivo (es decir, el usuario propietario del archivo será el usuario efectivo del proceso).

Set GID (SGID, `2000`): es lo mismo que SUID, pero al nivel de grupo (`g+s`), por lo que al ejecutar el archivo con SGID, el grupo propietario del archivo será el grupo efectivo del proceso. En un directorio, fuerza que sus nuevos ficheros usen su grupo.

> Usar los bits SUID y SGID presenta potenciales problemas de seguridad, por lo que por defecto no tienen efecto en scripts, aunque sí en binarios ejecutables.

Sticky bit (`1000`): es un bit (`+t`) especial que, al estar activo en un fichero o directorio, solo el usuario propietario puede eliminarlo o renombrarlo, aunque los demás tengan todos los permisos activos. Está presente en el directorio `/tmp`.

SUDO

Si un usuario cualquiera quiere ejecutar algún comando con permisos `root`, puede anteponer el comando **`sudo`** (*Super User DO*) al comando que se quiere ejecutar como `root`:

```
$ sudo [-u <usr>] <cmd> <args...>
```

Para poder usarlo, el usuario actual debe pertenecer al grupo `sudo` o estar en el archivo `/etc/sudoers`, dependiendo del SO

Antes de dar permiso, requiere introducir la contraseña del usuario actual, que después sigue recordando durante un rato:

```
$ sudo nano /etc/hosts
[sudo] password for adri:
```

Si se especifica el usuario y/o grupo con `-u <usuario>`/`-g <grupo>`, el comando se ejecutará como dicho usuario/grupo real:

```
$ sudo -u www-data mkdir /tmp/dir
[sudo] password for adri:
```

SU

Se puede abrir un **shell** bajo cualquier otro usuario si se conoce su contraseña o se tienen permisos `root`:

```
$ su [-] [<usuario>] [-s <shell>]
```

Si se pasa un nombre de usuario, inicia sesión como ese usuario. Si no, usa **`root`**.

Si se incluye `-` o la opción `-l`, crea un nuevo **shell con inicio de sesión** en lugar de abrir un subshell.

Si se usa `-s`, abre el intérprete shell especificado (ej: `/bin/bash`) para la sesión. Para usuarios sin shell, será necesario especificar una.

CHMOD

Cambia los permisos de uno o varios archivos:

```
$ chmod <permisos> <archivos...>
```

Los permisos pueden expresarse como los que se quieren conceder (+), establecer (=) o revocar (–) de entre los 3 clásicos: lectura (`r`), escritura (`w`) y ejecución (`x`). Se puede especificar si se quieren cambiar permisos de usuario (`u`), grupo (`g`) u otros (`o`).

También pueden expresarse como un número octal de 3 cifras donde la primera es de usuario, la segunda es de grupo y la tercera es de otros. Se puede añadir una 1ª cifra de permisos especiales

```
$ chmod +x script.sh
$ chmod go-r dir/*
$ chmod 600 passwords.txt
```

CHOWN

Cambia el usuario (y el grupo si se quiere) propietario de uno o más archivos:

```
$ chown [-LR] <usuario>[:<grupo>] <rutas...>
```

La opción `-R` (recursivo) hace que recorra también directorios si se especifican.

La opción `-L` permite atravesar enlaces simbólicos.

Ejemplo:

```
$ chown -R www-data:www-data /var/www
```

CHGRP

Cambia el grupo propietario de uno o más archivos:

```
$ chgrp [-LR] <grupo> <rutas...>
```

La opción `-R` (recursivo) hace que recorra también directorios si se especifican.

La opción `-L` permite atravesar enlaces simbólicos.

Ejemplo:

```
$ chgrp $USER dir/*
```

MÁSCARA DE PERMISOS

Los permisos originales de un archivo y un directorio recién creado son `0666` y `0777` respectivamente.

Sobre estos permisos se aplica una **máscara de permisos**, dando como resultado los permisos que posee un archivo o directorio cuando son creados.

Para determinar los permisos por defecto, se aplica un **AND** bit a bit entre la **máscara negada** (NOT bit a bit) y los **permisos originales** del fichero o directorio.

	Usuario			Grupo			Otros		
	r	w	x	r	w	x	r	w	x
Permisos originales de fichero (666)	1	1	0	1	1	0	1	1	0
Máscara 022 negada (755)	1	1	1	1	0	1	1	0	1
Permisos resultantes (666 & 755 = 644)	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

UMASK

Para **modificar** la **máscara de permisos**:

```
$ umask <máscara>
```

Para **ver** la **máscara de permisos**:

```
$ umask
```

La máscara por defecto en la mayoría de los sistemas tipo UNIX es `0022` o `0002`.

DESCRIPTORES DE FICHERO, CARACTERES DE CONTROL, REDIRECCIONES

COMANDOS ECHO, READ, TEE, CUT, TR, SORT, JQ

ENTRADA/SALIDA

ENTRADAS Y SALIDAS

En sistemas UNIX, cuando se abre un archivo para lectura/escritura se usa un **descriptor de fichero** (fd), un número entero que le asigna el sistema operativo que identifica el flujo de entrada/salida del fichero mientras está abierto.

Existen tres descriptores de fichero especiales:

- **Entrada estándar** o **stdin** (`0`): recibe bytes de la entrada del usuario por defecto. Interrumpe el flujo del programa hasta que el programa deja de pedir o hasta que recibe el carácter `EOF`.
- **Salida estándar** o **stdout** (`1`): vuelca bytes del resultado de la ejecución del programa en la terminal activa por defecto.
- **Salida de error** o **stderr** (`2`): vuelca bytes de errores, advertencias o información de debug en la terminal activa por defecto.

Estos descriptores especiales se pueden redirigir en archivos o entre ellos.

ECHO

Escribe los argumentos en la salida estándar:

```
$ echo <arg...>
```

Por defecto, `echo` imprime al final un salto de línea, pero podemos evitarlo con la opción `-n`.

`echo` no interpreta los caracteres de control. Para que los interprete correctamente, se usa la opción `-e`.

Ejemplos:

```
$ echo Hasta     aquí hemos  llegado
Hasta aquí hemos llegado
$ echo $PWD
/home/adri
$ echo "El directorio actual es $PWD"
El directorio actual es /home/adri
$ echo "\tas"
\tas
$ echo -e "\tas"
        as
```

CARACTERES DE CONTROL DEL SHELL

Código escape	Significado
`\a`	Sonido de alerta
`\b`	Retroceder (borrar) un carácter
`\c`	Suprimir la siguiente salida
`\f`	Cambiar de página
`\n`	Salto de línea
`\r`	Retorno de carro (inicio de línea)
`\t`	Tabulador horizontal
`\v`	Tabulador vertical
`\\`, `\$`, `\;`, etc. (para escapar caracteres especiales)	El mismo carácter (`\`, `$`, `;`, etc.), pero literal, sin interpretar
`\o`*`nnn`* (sh) | `\`*`nnn`* (bash)	Carácter ASCII con el código octal *`nnn`*

READ

Lee una línea de la entrada estándar:

```
$ read [-N <num>] <var...>
```

Los caracteres leídos (separados por espacios) los pone en las variables especificadas. La última variable se lleva el sobrante.

Si solo se especifica una variable, esa variable recibe toda la línea.

Muy útil para scripts que pidan datos al usuario por la entrada estándar.

Ejemplos:

```
$ read aaa bbb ccc
10 20 30 40 50 60
$ echo $aaa
10
$ echo $bbb
20
$ echo $ccc
30 40 50 60
```

LEER DE UN ARCHIVO

Se puede usar el contenido de un fichero como entrada estándar de un comando con el operador **<**:

```
$ <comando> < <fichero>
```

El archivo debe existir previamente.

Ejemplo:

```
$ cat < /etc/hosts
```

LEER UNA ENTRADA

Se puede pasar múltiples líneas por la entrada estándar de un comando de la siguiente forma:

```
$ <comando> <<DELIM
<texto multilínea...>
DELIM
```

Para detectar el final de la entrada, se define un **delimitador**. Se suele usar *EOF* (*End Of File*), pero se puede definir el que se quiera siempre y cuando la entrada termine en una línea que solo incluya dicho delimitador.

Ejemplo:

```
$ grep hola <<EOF
hola, amigos
Hola, amigo
Hello, everyone
Hola hola
EOF
```

Todo el texto se incluye en el historial del comando, por lo que no habrá que reescribirlo la próxima vez.

ESCRIBIR EN UN ARCHIVO

Se puede **redirigir la** salida estándar de un comando **a un fichero** con el operador **>**:

```
$ <comando> > <fichero>
```

Si el archivo no existe, se crea con ese contenido. Si existe, lo sobreescribe.

Ejemplo:

```
$ echo Contenido del archivo > newfile.txt
```

REDIRECCIONAR OTRA SALIDA

Si se quiere **redirigir una salida distinta** a la estándar, se puede especificar su **descriptor de fichero** (***n***) junto al operador de redirección:

```
$ <comando> n> <fichero>
```

Así, se pueden redirigir varias salidas a la vez:

```
$ rm dir1/ 1>f1 2>/dev/null
```

En este caso, se está redirigiendo la salida estándar al archivo `f1` y se está descartando la salida de error.

El operador `>`, sin descriptor al lado, asume que redirige la salida estándar, por lo que `>` es lo mismo que `1>`.

REDIRECCIÓN A OTRA SALIDA

Si se quiere **redirigir una salida a otra**, se usa el operador **>&** especificando ambos descriptores de fichero:

```
$ <comando> 2>&1 #stderr a stdout
$ <comando> 1>&2 #stdout a stderr
```

Se puede usar junto a otras redirecciones:

```
$ <comando> > <fichero1> 2>&1 > <fichero2>
```

En este caso, se está redirigiendo la salida de error a `<fichero1>` y la salida estándar a `<fichero2>`.

AGREGAR A UN ARCHIVO

El operador **>>** hace lo mismo que >, excepto que, en lugar de sobrescribir el archivo, **añade el texto al final** del mismo:

```
$ <comando> >> <fichero>  #Redirige la salida estándar
```

o

```
$ <comando> 2>> <fichero> #Redirige la salida de error
```

Ejemplo:

```
$ echo abc > append.txt
$ echo def >> append.txt
$ cat append.txt
abc
def
```

TUBERÍAS (PIPES)

Para **conectar la salida estándar** de un comando **a la entrada estándar** de otro comando:

```
$ <comando1> | <comando2>
```

La **tubería** (**|**) conecta la salida estándar de `<comando1>` con la entrada estándar de `<comando2>`. El resto de las entradas y salidas se mantienen igual.

Para redirigir ambas salidas (stdout y stderr) por una tubería:

```
$ <comando1> 2>&1 | <comando2>
```

Ejemplos:

```
$ grep sys /etc/passwd | cut -f 6 -d :
$ tr -dc '[:graph:]' </dev/random | head -c 32
```

FILTROS

Los siguientes comandos se usan a menudo con tuberías debido a que **leen de la** entrada estándar y **escriben en la** salida estándar:

- `tee`: escribe la entrada en un fichero.
- `grep`: busca patrones en la entrada.
- `head`/`tail`: extrae las primeras/últimas líneas o bytes de la entrada.
- `sed`: filtra y transforma texto de un flujo.
- `awk`: escanea patrones y procesa el texto.
- `cut`: selecciona una parte de cada línea.
- `wc`: cuenta las líneas, palabras y bytes del texto. Con `-l`, `-w` y `-c`, selecciona cada dato respectivamente.
- `tr`: reemplaza caracteres en el texto.
- `sort`: ordena las líneas.
- `fold`: ajusta las líneas a un ancho.
- `tac`/`rev`: invierte las líneas/letras.
- `jq`: procesa JSON.
- `base64`: codifica en base64. Con `-d`, decodifica.

TEE

Lo que recibe de la entrada estándar lo escribe en archivo(s) y en la salida estándar:

```
$ tee [-a] <archivo...>
```

Si los archivos no existen, los crea. Si existen, los sobreescribe.

Con la opción `-a` no los sobreescribe, sino que se añade el nuevo contenido al final.

Ejemplo:

```
# cat /etc/shadow | grep $USER | tee passwd.txt >/dev/null
```

CUT

Corta cada línea de la entrada estándar (o de ficheros), según las opciones que reciba:

```
$ cut <opciones> [<ficheros...>]
```

Se puede usar de una de las siguientes formas:

- Selecciona cierta posición de bytes (`-b <pos>`) o caracteres (`-c <pos>`).
- Selecciona con respecto a un delimitador mediante `-f <pos>` (el delimitador por defecto es el tabulador, pero se puede especificar otro con `-d <delim>`).

`<pos>` es un número (`N`), un rango entre dos números (`N-M`), desde un número hasta el final (`N-`) o desde el principio hasta un número (`-M`).

Se puede invertir una selección con la opción `--complement`.

TR

Traduce o elimina caracteres de la entrada estándar y lo envía a la salida estándar:

```
$ tr [<opciones>] <caracteres1> [<caracteres2>]
```

`tr` sustituye cada ocurrencia de un carácter de `<caracteres1>` por el carácter de `<caracteres2>` que tiene la misma posición.

`<caracteres1>` y `<caracteres2>` definen vectores de caracteres. Se pueden definir directamente (`abcde123_=`), mediante rangos (`a-z0-9`) y/o mediante vectores predefinidos (`[:alnum:]`) [ver expresiones entre corchetes].

Con `-d`, borra las ocurrencias de `<caracteres1>` en lugar de traducirlas.

Con `-c`, traduce o borra los caracteres que NO están en `<caracteres1>`.

SORT

Ordena las líneas de la entrada estándar (o de ficheros):

```
$ sort [<ficheros...>]
```

Por defecto ordena según el código ASCII de los caracteres del texto. Algunas opciones:

- `-b` para ignorar los primeros espacios.
- `-d` para considerar solo letras, números y espacios.
- `-f` para no distinguir entre mayúsculas y minúsculas.
- Criterio de ordenación:
  - `-n` para números.
  - `-R` para orden aleatorio.
  - `-V` para versiones.
- `-r` para invertir la ordenación.
- `-k CAMPO[.POS][,CAMPO[.POS]]` para ordenar según una clave.
- `-t <delim>` para definir un delimitador (para los campos en `-k`).
- `-u` para solo imprimir valores únicos.

JQ

Procesa datos de la entrada estándar (o de ficheros) en formato JSON y devuelve un resultado (JSON) por la salida estándar. Uso:

```
$ jq <filtro> [<ficheros...>]
```

Algunas opciones son:

- `-R` (`--raw-input`) para que interprete la entrada como texto, no JSON.
- `-r` (`--raw-output`) para que la salida sea texto, no en formato JSON.

Ejemplos:

```
$ echo '{"foo": [5,"bar","baz",null]}' | jq -n '.foo|.[2]'
$ echo 'dc=example,dc=com' | jq -Rr @uri
```

EXPRESIONES REGULARES

COMANDOS GREP Y SED

EXPRESIONES REGULARES Y EDICIÓN DE FLUJO

EXPRESIONES REGULARES

Una **expresión regular** es un patrón que describe un conjunto de cadenas de caracteres. Es una herramienta muy poderosa para buscar patrones en textos.

Una expresión regular la forman **caracteres** (letras, números, símbolos, espacios, etc.) concatenados entre sí, posiblemente acompañados de operadores.

> **Un texto es la expresión regular más básica**: `"hola, amigo"` es una expresión regular que selecciona cualquier aparición del texto `hola, amigo` dentro de ella (en ese orden e incluyendo la coma y el espacio).

A continuación, se describen componentes que nos permiten hacer potentes búsquedas de patrones mediante las **expresiones regulares extendidas** (opción `-E` en `grep` y `sed`).

EXPRESIONES REGULARES

- El **punto** (**.**) representa cualquier carácter (menos el salto de línea).
- **Expresiones entre corchetes**: para seleccionar un carácter de entre una lista de caracteres, se encierran dichos caracteres entre corchetes.
  - Se pueden definir **rangos**, escribiendo un guión (–) entre dos caracteres (`a-z`) para indicar cualquier carácter cuyo código ASCII esté entre el primer carácter (`a`) y el segundo (`z`).
  - Una expresión entre corchetes puede **negarse** poniendo `^` al comienzo de la expresión (`[^abc]`), con lo que se seleccionaría cualquier carácter excepto los que aparecen tras el `^` (`abc`).
  - Existen clases de caracteres con nombre, como `[:alnum:]`, `[:alpha:]`, `[:blank:]`, `[:space:]`, `[:graph:]`, `[:punct:]`, etc. Deben ir entre los corchetes exteriores.

  Ejemplos: `[aeiou]`, `[a-zA-Z0-9_]`, `[^/]`, `[[:graph:]]`, `[_[:alnum:]]`.

EXPRESIONES REGULARES

- **Anclajes**: los caracteres `^` y `$` representan respectivamente el **comienzo** y el **final** de la línea (o del texto).
- **Límites de palabra**: existen expresiones especiales para seleccionar los límites de una palabra: límite inicial (`\<`), límite final (`\>`) o cualquiera de ellos (`\b`).
- **Caracteres especiales**: los caracteres `.`, `?`, `*`, `+`, `(`, `)`, `[`, `]`, `{`, `}`, `^`, `$`, `|` y `\` (y – dentro de una expresión entre corchetes) son caracteres especiales en una expresión regular, por lo que, si se quieren incluir como caracteres literales en una expresión, a menudo será necesario escaparlos anteponiendo un `\`.

  Por ejemplo: `\.`, `\(`, `\\`, etc.

EXPRESIONES REGULARES

- **Repeticiones**: para indicar una cierta repetición de una expresión precedente:
  - `?` indica 0 o 1 repetición (es decir, una expresión opcional).
  - `*` indica 0 o más repeticiones.
  - `+` indica 1 o más repeticiones.
  - `{n}` indica exactamente `n` repeticiones.
  - `{n,}` indica `n` o más repeticiones.
  - `{,m}` indica `m` o menos repeticiones.
  - `{n,m}` indica entre `n` y `m` repeticiones (ambos inclusive).

  Ejemplos:
  - `ca+r` selecciona los textos: `car`, `caar`, `caaar`, `caaaar`, `caaaaar`, etc.
  - `0b([01]){2}` selecciona los siguientes textos: `0b00`, `0b01`, `0b10` y `0b11`.

EXPRESIONES REGULARES

- **Concatenación**: al poner una expresión después de otra (concatenadas), coincidirá con cualquier cadena formada por una subcadena que concuerde con la primera expresión seguida de otra subcadena que concuerde con la segunda expresión.
- **Alternativas**: al separar dos expresiones mediante `|`, la expresión resultante coincidirá con cualquier cadena que concuerde con cualquiera de las dos expresiones.
- **Grupos**: se puede encerrar una expresión entre `(` y `)` para crear un grupo.
  - Los grupos se pueden usar para aplicar repeticiones a expresiones compuestas o bien para referenciarlos después como `\N`, donde `N` es un dígito entre el `1` y el `9` que indica el orden del grupo en la expresión regular (por ejemplo, `\2` será el segundo grupo).

GREP

Busca patrones (**expresiones regulares**) en el contenido de archivos (o stdin si no se pasan ficheros) y muestra las líneas que los cumplen por stdout:

```
$ grep <regexp> [<ficheros...>]
```

El estado de salida o código de error de `grep` será:

- `0` si encuentra algún resultado.
- `1` si no encuentra ningún resultado.
- `2` si ha ocurrido algún error.

GREP - OPCIONES

Algunas opciones de `grep`:

- `-i` (ignore case) para no distinguir entre mayúsculas y minúsculas
- `-c` (count) para que solo muestre el número de líneas que coinciden
- `-r` (recursive) para hacer búsqueda recursiva en directorios
- `-v` (invert match) para mostrar solo las líneas que no coinciden con el patrón
- `-o` (only matching) para mostrar solo los fragmentos que coinciden con el patrón.
- `-q` (quiet) para no mostrar salida estándar, solo se espera ver el código de error.
- `-E` (extended) para usar expresiones regulares extendidas en lugar de las básicas

SED

El comando `sed` es un editor de flujo que nos permite hacer transformaciones básicas de la entrada estándar (o el contenido de archivos si se le pasa más de un argumento):

```
$ sed <script-sed> [<archivos...>]
```

El primer argumento es un **script sed** para realizar las transformaciones. Opciones:

- `-E` para usar expresiones regulares extendidas en lugar de las básicas.
- `-n`: deshabilita imprimir (o escribir, si se usa con `-i`) el texto automáticamente.
- `-i[SUFIJO]`: modifica los ficheros, en lugar de solo sacar la salida modificada. Si se especifica `SUFIJO`, creará un backup con ese sufijo.
- `-s`: procesa los archivos de forma separada, y no como un único flujo de entrada.

SCRIPTS SED

Un **script sed** está formado por una serie de **comandos sed separados por `;`**.

Un **comando sed** sigue la siguiente sintaxis:

```
[lin][!]X[opciones]
```

Donde:

- `X` es un comando sed de una letra.
- Si se especifica `lin`, el comando se ejecutará solo en esa(s) línea(s). `lin` puede ser un número (`36`), el carácter `$` (última línea), una expresión regular (`/[a-z]+/`) o un rango entre cualesquiera dos de las anteriores (`2,/^#/`). Si se incluye `!`, el comando se ejecutará en todas las líneas menos en las seleccionadas por `lin`.
- Se especifican `opciones` para ciertos comandos sed.

También se pueden agrupar comandos con `{` y `}`:

```
[lin][!]{ comandos }
```

FLUJOS Y BUFFERS EN SED

Para trabajar con `sed` es necesario comprender los siguientes conceptos:

- **Flujo de entrada**: es el texto original, antes de ser modificado.
- **Flujo de salida**: es el texto resultante, después de ser modificado.
- ***Pattern space*** : es el buffer que contiene la línea actual en cada momento.
- ***Hold space*** : es un buffer "portapapeles" en `sed`. Lo usan `h`, `H`, `g`, `G` y `x`.

`sed` empieza cargando la primera línea del flujo de entrada en el espacio patrón.

Los comandos pueden modificar el pattern space. Cuando no hay nada más que hacer en la línea actual, `sed` imprime el pattern space en el flujo de salida (salvo con la opción `-n`), borra el pattern space y carga en él la siguiente línea del flujo de entrada.

COMANDOS SED

Principales **comandos sed**:

- `s/REXP/REEMPL/[FLAGS]`: el comando más conocido de `sed`. Si hay una coincidencia con la expresión regular `REXP` en el espacio patrón, la sustituye por `REEMPL`.

  El carácter que sigue a `s` es el delimitador (no necesariamente `/`), preferiblemente un carácter que no esté en la expresión.

  Por ejemplo, `s:/usr/lib:/lib:g` es lo mismo que `s|/usr/lib|/lib|g`, etc.

- `n`: imprime el espacio patrón (salvo con la opción `-n`) y carga la siguiente línea.
- `d`: no imprime el espacio patrón y carga la siguiente línea.
- `p`: imprime el espacio patrón (se suele usar junto con la opción `-n`).

OTROS COMANDOS SED

Comandos sed que manipulan el *hold space*:

- `h`: "copia" el espacio patrón en el *hold space*. Si ya había contenido, lo sustituye.
- `H`: "copia" el espacio patrón en el *hold space*. Si ya había contenido, lo añade.
- `g`: sustituye el contenido del espacio patrón por el del *hold space*.
- `G`: añade el contenido del *hold space* al final del espacio patrón (en una nueva línea).
- x: intercambia el contenido del espacio patrón con el del *hold space*.

OTROS COMANDOS SED

Otros comandos sed:

- `aTEXTO`: añade una nueva línea con `TEXTO` después de la línea actual.
- `iTEXTO`: añade una nueva línea con `TEXTO` antes de la línea actual.
- `cTEXTO`: reemplaza la línea actual por `TEXTO`.
- `r FICH`: lee y añade el contenido del fichero `FICH` en el flujo de salida.
- `w FICH`: escribe el espacio patrón en `FICH`.
- `z`: vacía el contenido del espacio patrón.
- `=`: imprime el número de la línea actual.

OTROS COMANDOS SED

Etiquetas, saltos y salida:

- `: ETIQ`: crea una etiqueta a la que poder saltar.
- `b ETIQ`: salta incondicionalmente a una etiqueta.
- `t ETIQ`: salta a una etiqueta solo si se ha llevado a cabo con éxito una sustitución (`s`) en la línea actual.
- `T ETIQ`: salta a una etiqueta solo si NO se ha llevado a cabo con éxito ninguna sustitución (`s`) en la línea actual.
- `q[COD]`: sale de `sed` sin procesar más comandos, opcionalmente con un código de error (`COD`).

SUSTITUCIÓN SED

En el comando `s`, el texto de reemplazo (`REEMPL`) puede incluir:

- `&`: se sustituye por el texto seleccionado. Ejemplo:

```
$ echo 'abc 123 def' | sed -E 's/[0-9]+/&=&/'
abc 123=123 def
```

- `\`*N*: se sustituye por el *N*-ésimo grupo de la expresión regular en el texto seleccionado. Por ejemplo:

```
$ echo '1@2, aa@ff' | sed -E 's/([a-z]+)@([a-z]+)/\2@\1/i'
1@2, ff@aa
```

- `\L` o `\U`: pone los siguientes caracteres en minúscula (`\L`) o mayúscula (`\U`) hasta encontrar `\E`. También se puede cambiar solo el siguiente carácter (`\l` o `\u`).

SUSTITUCIÓN SED

Por defecto, una sustitución solo se aplica a la primera coincidencia del espacio patrón.

Se pueden incluir algunas **banderas** (`FLAGS`) al final de `s` para modificar su comportamiento:

- `g` (global): busca y sustituye todas las coincidencias que encuentre, no solo la 1ª.
- *N*: solo reemplaza la coincidencia número *N*.
- `i` (ignore case): no diferencia entre mayúsculas y minúsculas.
- `p`: si se hizo la sustitución, imprime la línea modificada (usar junto con la opción `-n`).
- `w FICH`: escribe la línea modificada en `FICH`.

EJEMPLOS SED

Ejemplo de los comandos `p`, `n` y `s`:

```
$ sed '1,5p ; /^foo/!n ; s/hello/world/g' input.txt
```

Ejemplo de agrupamientos (eliminar comentarios y líneas vacías de un script shell):

```
$ sed -f - script.sh <<EOL
2,$ {           #Entre la segunda y la última línea:
  /^#/d         #  Eliminar comentarios (empiezan por #)
  2! { /^$/d } #  Eliminar líneas vacías salvo la segunda
}
EOL
```

ESTADOS DE UN PROCESO, PROCESOS EN PRIMER PLANO Y SEGUNDO PLANO, SEÑALES, TAREAS PROGRAMADAS

COMANDOS PS, TOP, KILL, JOBS, FG, BG, WAIT, NICE, CRONTAB

PROCESOS

PROCESOS

Un **proceso** o tarea es una instancia de un programa en ejecución. Los procesos tienen un contexto, como su process-ID (**PID**), procesos hijos, proceso padre, los recursos que está consumiendo, los permisos que tiene, etc.

Un proceso tiene información de dos tipos de usuarios y grupos:

- **El usuario y grupo real** es el que ha ejecutado el proceso.
- **El usuario y grupo efectivo** suele ser igual que el real, pero puede ser diferente si se ejecutó un archivo con el bit set-user-ID (SUID) o set-group-ID (SGID) activo, en cuyo caso se ejecutaría bajo los privilegios del usuario o grupo propietario del archivo, aunque el que inició el proceso fuera otro usuario diferente.

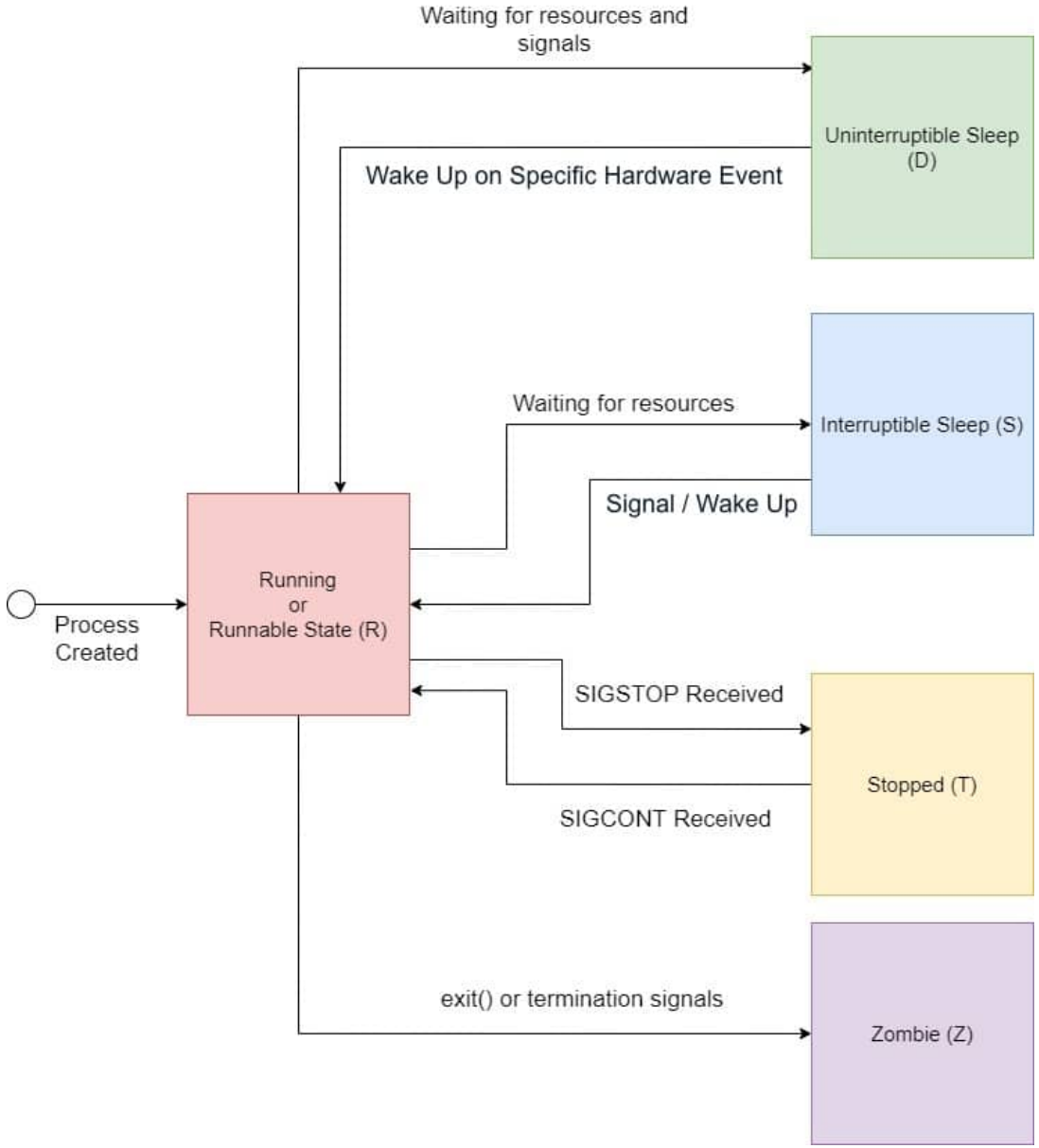
ESTADOS DE UN PROCESO

Los principales estados de un proceso en Linux son:

- **Espera no interrumpible (`D`)**: el proceso espera por una operación de entrada/salida.
- **Espera interrumpible (`S`)**: el proceso espera a algún evento para su ejecución.
- **En ejecución (`R`)**: el proceso está ejecutándose (con tiempo en la CPU o en la cola).
- **Detenido (`T`)**: el proceso está detenido por el envío de una señal generalmente.
- **Zombie (`Z`)**: el proceso terminó, pero su proceso padre, que sigue vivo, no lo sabe.

La letra de cada estado es la que muestra el comando `ps` al ver el estado de los procesos.

ESTADOS DE UN PROCESO



PS

Muestra los procesos activos (junto a sus PID) de este shell:

```
$ ps [<opciones>] [<pid>]
```

Con la opción `-e`, muestra todos los procesos del sistema.

Para mostrar los procesos en forma de árbol, se usa `-ejH` (o se usa el comando `pstree`).

Para mostrar información sobre los hilos, se usa `-eLf`.

Para ver los procesos en un formato orientado a usuario, se agrega el argumento `u`.

Para filtrar los procesos que se ejecutan como un usuario, se usa `-U <usuario>` (real) y/o `-u <usuario>` (efectivo). Para grupos, `-G <grupo>` y/o `-g <grupo>`. Ejemplo:

```
ps -U root -u root -j
```

TOP

Este comando muestra todos los procesos que hay en el sistema, junto a algunas estadísticas de rendimiento, en tiempo real (se refresca cada 3 segundos):

```
$ top
```

La lista se puede recorrer como un archivo abierto con `less`. Se sale con `q`.

Existen alternativas más visuales e interactivas como `htop` o `btop`.

Hay también alternativas muestran otro tipo de información, como de red (`iftop`), de entrada/salida (`iotop`) o herramientas más avanzadas de monitorización, como `csysdig` o `perf`.

KILL

El built-in `kill` envía una señal a uno o varios procesos. Se suele usar para matar procesos:

```
$ kill [-<señal>] <pid...>
```

La señal enviada por defecto es `SIGTERM`. Si se quiere enviar otra distinta, se puede especificar mediante su nombre o número.

Si se quiere forzar a terminar el proceso con PID 15972, ejecutamos cualquiera de las siguientes líneas:

```
$ kill -SIGKILL 15972 #Por nombre largo
$ kill -KILL 15972    #Por nombre
$ kill -9 15972       #Por código
```

SEÑALES

En Linux, una **señal** es un código que se envía en relación con un proceso con un fin determinado (detenerlo, matarlo, etc.).

- En general, cuando se envía una señal a un proceso, este decide qué hacer.

  Puede realizar la acción por defecto (terminar o detenerse), realizar alguna tarea alternativa o ignorar la señal.

- **Las señales `SIGKILL` y `SIGSTOP` no se pueden ignorar**.

SIG	#	Descripción
`HUP`	1	Colgar (*hang up*). Al cerrar una terminal, se envía esta señal a sus subprocesos para terminarlos
`INT`	2	Interrumpir un proceso (`Ctrl+C`). Suele implicar la terminación del proceso
`KILL`	9	Enviado al SO para forzar la terminación de un proceso
`TERM`	15	Terminar un proceso
`CONT`	18	Enviado al SO para continuar un proceso detenido
`STOP`	19	Enviado al SO para detener un proceso
`TSTP`	20	Detener un proceso (`Ctrl+Z`)

TAREAS EN SEGUNDO PLANO

Por defecto, todos los comandos que ejecutamos en el shell lanzan **procesos en primer plano**, es decir, procesos que paralizan el shell (no se pueden ejecutar más comandos) hasta que terminan o reciben una interrupción.

Un **proceso en segundo plano** o **trabajo**, en cambio, no interfiere con el funcionamiento del shell durante su ejecución.

Se puede lanzar un proceso en segundo plano usando el operador `&`:

```
$ <comando> &
```

Un proceso en primer plano en ejecución se puede detener y mandar a segundo plano con `Ctrl+Z` (señal `SIGTSTP`)

JOBS

El built-in `jobs` muestra una lista de trabajos activos y detenidos:

```
$ jobs
```

Los **trabajos** son procesos en segundo plano, activos o detenidos, que tienen un identificador propio (job ID), que aparece como `[n]`, distinto al PID, y que al referenciarlo se escribe como `%n`. Ejemplo:

```
$ sleep 1000 &
$ jobs
[1]+  Running                 sleep 1000 &
```

En comandos que reciben PID, como `kill`, se pueden especificar job ID en lugar de PID.

GESTIONAR TRABAJOS

Para **detener un proceso en primer plano**, se usa `Ctrl+Z` (señal `SIGTSTP`).

Para **detener un trabajo**, se puede usar el comando `kill` con la señal `SIGTSTP`:

```
$ kill -TSTP %<job-id>
```

Para **reanudar en segundo plano** un trabajo detenido, se usa el built-in **`bg`**:

```
$ bg %<job-id>
```

Para **traer (o reanudar) a primer plano** un trabajo, se usa el built-in **`fg`**:

```
$ fg %<job-id>
```

Para bloquear el shell hasta que uno o varios procesos terminen de ejecutarse:

```
$ wait <pid/job-id...>
```

NOHUP

Cuando una terminal se cierra antes de terminar algún proceso, se envía una señal `SIGHUP` a estos procesos que aún cuelgan de ella para terminarlos.

Para lanzar un proceso inmune a la señal `SIGHUP` (es decir, que la ignore):

```
$ nohup <comando> <argumentos...>
```

Si la entrada estándar del comando es la terminal, la redirigirá a un fichero ilegible.

Si la salida estándar y/o de error del comando son la terminal, las redirigirá al fichero `nohup.out`.

Al cerrar la terminal que lo lanzó, el proceso dejará de estar asociado con la terminal y será adoptado por el proceso `init`.

PRIORIDADES

Todos los procesos en Linux se ejecutan con una determinada **prioridad**, un número entero entre `-20` (la más favorable) y `19` (la menos favorable). La prioridad predeterminada para procesos de usuario es `0`.

Para ejecutar un proceso con una determinada prioridad:

```
$ nice -n <prioridad> <comando>
```

Por ejemplo:

```
$ nice -n 18 python3 tarea_pesada.py
```

El comando `ps` con la opción `-l` muestra los valores `nice` en la columna `NI`.

CRON

Cron es una poderosa utilidad para programar y automatizar tareas en sistemas UNIX, tanto horarias, diarias, semanales, mensuales o anuales.

Para añadir o editar tareas programadas del usuario actual (la primera vez preguntará por el editor preferido):

```
$ crontab -e
```

Para mostrar las tareas programadas del usuario actual:

```
$ crontab -l
```

Cada usuario tiene un fichero cron, por lo que no se mostrará lo mismo al ejecutar `crontab` desde un usuario que desde otro.

TAREAS DE CRON

Una **tarea de cron** tiene el siguiente aspecto:

```
0 5 * * 0 sh /ruta/backup.sh
```

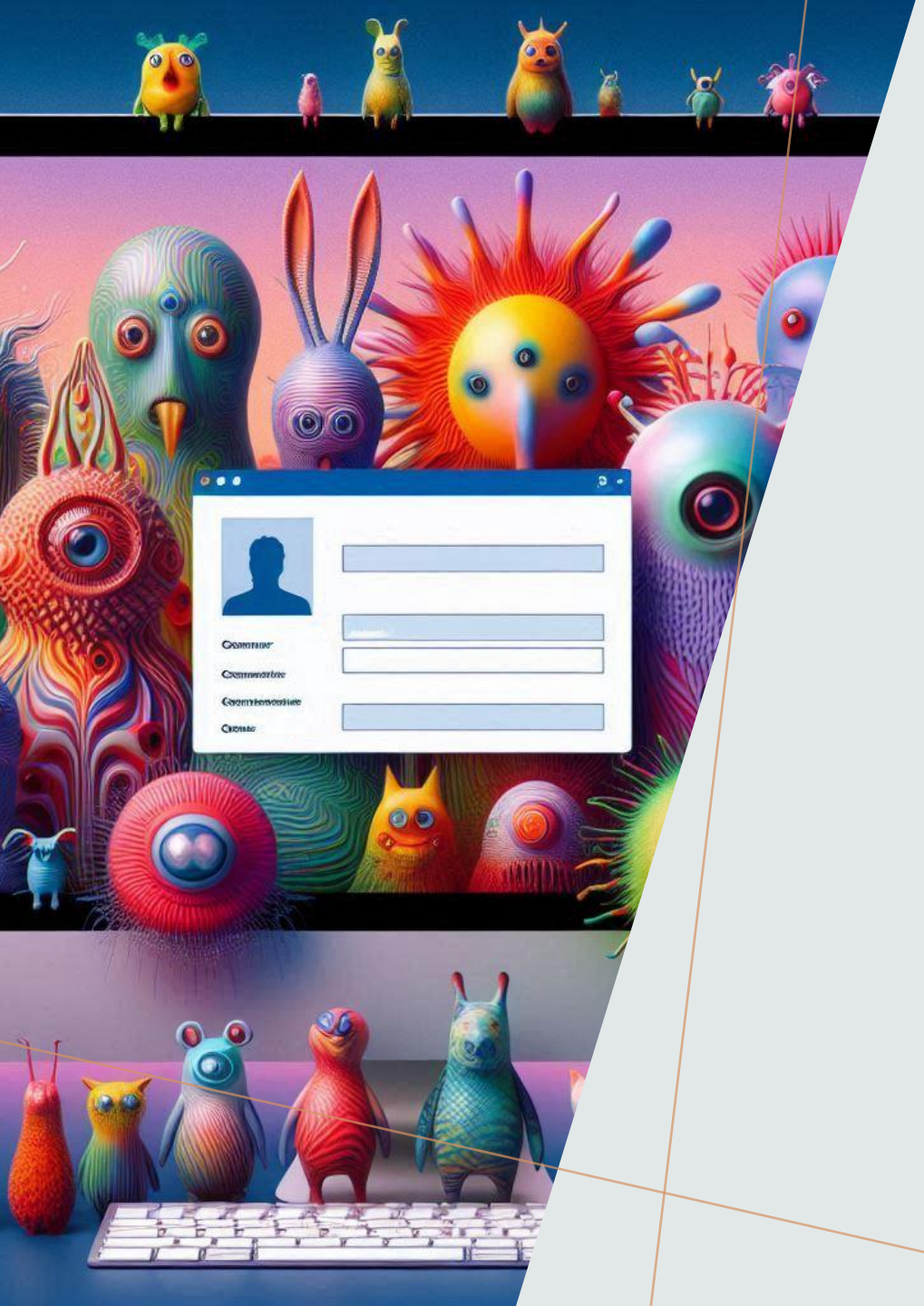
También se pueden usar palabras reservadas de la siguiente forma:

```
@reboot date >/tmp/dates.log
```

Cron permite redireccionar la salida del comando.

Cuando un comando de cron muestra algo por la salida estándar o de error, la envía por correo al usuario (si tiene MTA).

#	Significado	Rango de valores	Valores
1	Minuto	0-59	***** para todos los valores **?** para cualquier valor (solo día del mes o día de la semana) **n** para un valor concreto **n-m** para un rango **v1,v2** para una lista **v1/v2** desde v1, cada v2 (v1, v1+v2, v1+2*v2, etc.)
2	Hora	0-23	
3	Día (mes)	1-31	
4	Mes	1-12	
5	Día (semana)	0-6 SUN-SAT	
6	Comando	Cualquier comando del shell	



USUARIOS Y GRUPOS EN LINUX

COMANDOS LOGIN, ID, WHO, ADDUSER, ADDGROUP, DELUSER, DELGROUP, PASSWD

GESTIÓN DE USUARIOS Y GRUPOS

USUARIOS

En Linux, cada usuario tiene un **nombre de usuario**, un **UID** (identificador de usuario) y un **GID** (el identificador de su **grupo principal**).

Un usuario normal suele tener algunos datos adicionales como nombre completo, teléfono del trabajo, etc. y, además, una **carpeta personal**, bandejas de correo y una **contraseña** para iniciar sesión.

En todo sistema UNIX existe un **superusuario** o **usuario administrador** (`root`) que tiene todos los privilegios del sistema y acceso total a los ficheros y directorios. Su directorio personal es `/root`, su UID es `0` y su GID es `0`.

TIPOS DE USUARIOS

En Linux existen varias clases de usuarios:

- **Usuarios normales**: pueden iniciar sesión, pertenecen a un grupo principal, tienen funcionalidad limitada y acceso limitado a ficheros y directorios, pero acceso total a su propio directorio personal (por defecto en `/home/<usuario>`).
- **Usuarios de sistema**: suelen crearse para ejecutar servicios específicos en su nombre, con lo que asumen distintos privilegios del sistema. Algunos no pueden iniciar sesión, no tienen grupo principal propio y/o no tienen directorio personal.

A los usuarios normales y de sistema se les asignan UID de rangos diferentes, según la configuración del sistema.

GRUPOS

Un usuario siempre tiene un **grupo principal**, pero puede pertenecer a más grupos también (**grupos secundarios**).

En Linux, un grupo consiste en un **nombre de grupo** y un **GID** (identificador de grupo).

Un grupo puede contener cualquier número de usuarios.

Al igual que los usuarios, existen los **grupos normales** y los **grupos de sistema**, que se diferencian en el rango de GID asignados.

CONFIGURACIÓN DE USUARIOS

Los siguientes son archivos de configuración de usuarios y grupos:

- `/etc/passwd`: almacena los datos básicos de los usuarios del sistema operativo:

  Nombre, contraseña sin cifrar, UID, GID, datos adicionales, directorio personal e intérprete del shell.

- `/etc/shadow`: guarda las contraseñas cifradas de los usuarios, además de las configuraciones de caducidad. Para verlo se necesitan permisos de `root`.

- `/etc/group`: guarda la información básica de los grupos del sistema operativo:
  - Nombre, contraseña sin cifrar, GID y lista de usuarios miembros.

DIRECTORIO PERSONAL

Cada usuario tiene un directorio personal configurado en `/etc/passwd`.

- El directorio personal de `root` es `/root`.
- Los usuarios normales tienen, por defecto, un directorio personal propio dentro de `/home`.
- Los usuarios de sistema no suelen usar directorios personales, por lo que a menudo se les asigna directorios como el raíz (`/`) o directorios inexistentes.

Al crear nuevos usuarios, existen opciones para crear (o no) su directorio personal, copiando el contenido del directorio `/etc/skel` (el esqueleto de los nuevos directorios personales) en él.

LOGIN

Para **iniciar una nueva sesión** de un usuario:

```
$ login [<usuario>]
```

Esto nos pedirá un nombre de usuario (salvo que se especifique el usuario en el primer argumento) y la contraseña del usuario.

Si se usa la opción `-f`, se omitirá la autenticación (solo puede hacerlo `root`):

```
# login -f <usuario>
```

Si el inicio de sesión se realiza con éxito, se abrirá un nuevo shell de inicio de sesión con el usuario especificado.

Este es el comando que usa el sistema (y SSH) para iniciar una sesión.

`login` no equivale a `su -`.

ID

Para obtener el UID, GID (grupo principal) y grupos a los que pertenece el usuario (efectivo) actual o un usuario determinado:

```
$ id
$ id <usuario>
```

Con la opción `-u`, muestra solo el usuario (su UID).

Con la opción `-g`, muestra solo su grupo principal (su GID).

Con la opción `-G`, muestra solo los grupos a los que pertenece (sus GID).

Con cualquiera de las 3 opciones anteriores, la opción `-n` muestra nombre(s) en lugar de IDs.

Con `-r`, muestra la información del usuario real actual, no del efectivo.

WHO

Devuelve las sesiones abiertas en el sistema. Para cada sesión muestra el usuario conectado, la terminal virtual (TTY) o emulada (PTS/) abierta, la fecha en la que se inició la sesión y, si es en remoto (SSH), la dirección IP del cliente.

```
$ who [<opciones>]
```

Con la opción `-m`, devuelve la sesión actual. Equivalente a:

```
$ who am i
```

Con la opción `-l`, muestra los procesos de login activos del sistema.

El comando `w` muestra información parecida a `who`, además del proceso líder de cada sesión.

OBTENER INFORMACIÓN DE SESIONES

Otros comandos para obtener información de la sesión actual:

- `whoami`: devuelve el nombre del usuario (efectivo) actual, al igual que `$USER`.
- `groups`: muestra los grupos del usuario (efectivo) actual (igual que `id -Gn`).
- `tty`: muestra la ruta de la terminal usada por la sesión actual.
- `last`/`lastb`: muestra una lista de los últimos inicios de sesión exitosos/fallidos en el sistema.
- `lastlog`: muestra la lista de usuarios y la información de su última sesión.

Además, están a disposición del usuario las variables de entorno `USER`, `PATH`, `HOME`, `SHELL` y `TERM`.

CREAR UN USUARIO

Para crear un usuario de forma interactiva:

```
# adduser <usuario>
```

Con la opción `--system`, se crea un **usuario de sistema**, sin carpeta personal, sin contraseña y sin grupo propio (salvo que se especifique mediante opciones).

Si se crea un **usuario normal**, el comando crea un grupo con el mismo nombre, se lo asigna como grupo principal, crea su directorio personal en `/home` y pide su nueva contraseña y datos adicionales.

CREAR UN GRUPO

Para crear un grupo:

```
# addgroup <grupo>
```

Con la opción `--system`, se crea un grupo de sistema.

Para añadir un usuario a un grupo:

```
# adduser <usuario> <grupo>
```

ELIMINAR USUARIOS Y GRUPOS

Para borrar un usuario:

```
# deluser <usuario>
```

Este comando no borra el directorio personal ni las bandejas de correo, salvo si se incluye la opción `--remove-home`.

Para borrar un grupo:

```
# delgroup <grupo>
```

Para quitar a un usuario de un grupo:

```
# deluser <usuario> <grupo>
```

OPERACIONES CON USUARIOS Y GRUPOS DE BAJO NIVEL

Existe un conjunto de **comandos de bajo nivel** que realizan las tareas de añadir (`useradd`, `groupadd`) y borrar (`userdel`, `groupdel`) usuarios y grupos. Suelen resultar adecuados para usar dentro de scripts.

En cambio, a un usuario le resultará más cómodo invocar utilidades de más **alto nivel** (`adduser`, `addgroup`, `deluser`, `delgroup`) que realizan por defecto otras acciones útiles como la creación automática del directorio personal o la introducción de datos del usuario de forma interactiva.

También se pueden **modificar usuarios y grupos** mediante los comandos `usermod` y `groupmod` respectivamente.

CAMBIAR CONTRASEÑA

Para iniciar una sesión con un usuario, este necesita tener una contraseña.

Para cambiar (o establecer) la contraseña del usuario actual:

```
$ passwd
```

Si se quiere cambiar (o establecer) la de cualquier usuario, se necesitan permisos de administrador:

```
# passwd <usuario>
```

`passwd` pide la contraseña de forma interactiva. Puede borrarse con `--delete`.

El comando `chpasswd` realiza la misma tarea, pero de forma no interactiva (lee de la entrada estándar líneas de la forma `<usuario>:<contraseña>`).

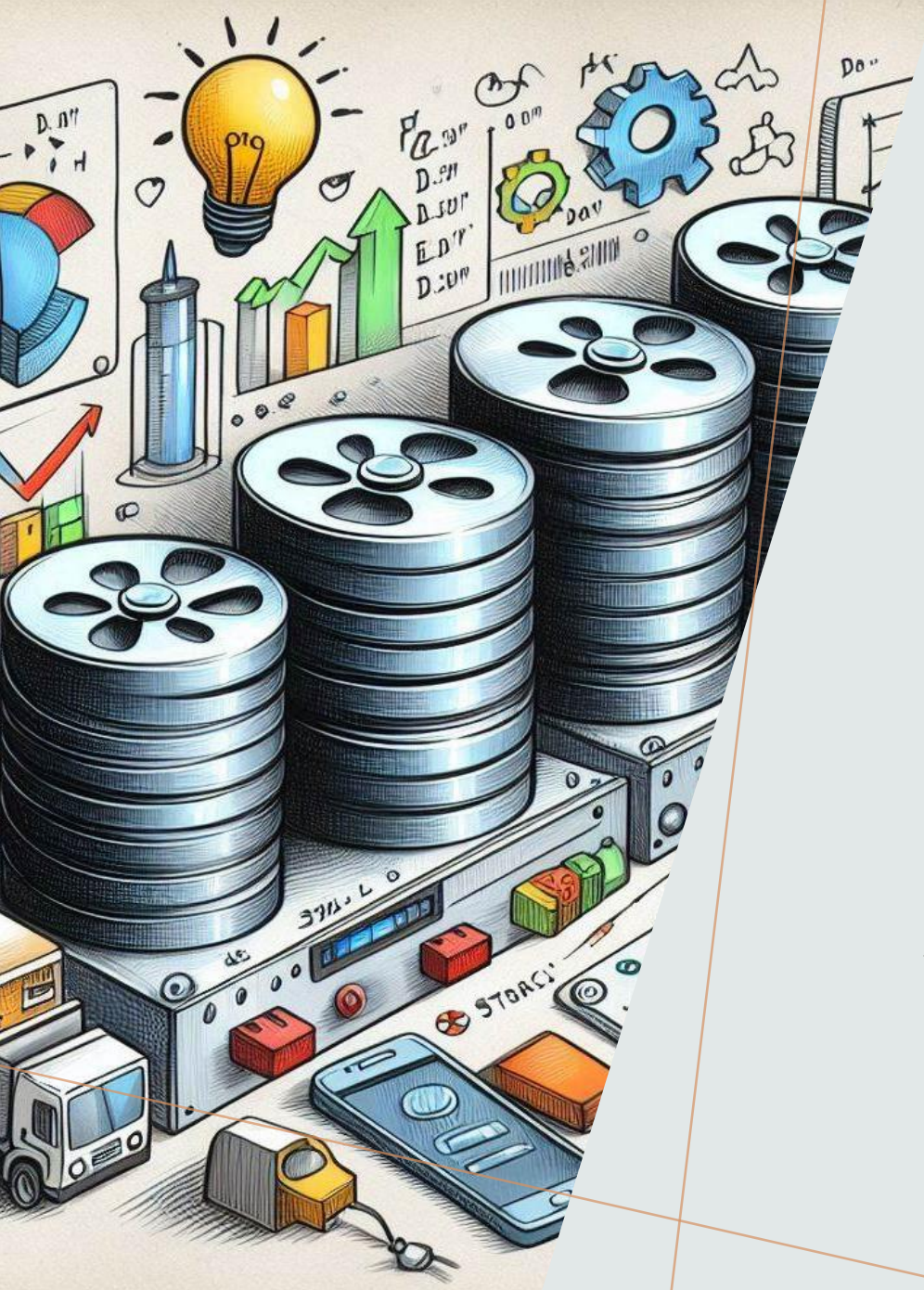
CADUCIDAD DE CONTRASEÑAS

Para establecer la caducidad de la contraseña de un usuario:

```
# chage <usuario> <opciones...>
```

Estas configuraciones se guardan en `/etc/shadow`. Algunas opciones:

- `-l` muestra la información de caducidad del usuario.
- `-d <fecha>` cambia la fecha de último cambio de contraseña.
- `-E <fecha>` cambia la fecha de caducidad de la cuenta (después no deja iniciar sesión).
- `-m <n>` cambia el número mínimo de días para el cambio de contraseña desde el anterior.
- `-M <n>` cambia el número máximo de días en que su contraseña puede estar en vigor.
- `-W <n>` muestra una advertencia los últimos `<n>` días de caducidad de la contraseña.



DISCOS, PARTICIONES Y SISTEMAS DE ARCHIVOS

COMANDOS FDISK, MKFS, MOUNT, UMOUNT, LSBLK, DF

ADMINISTRACIÓN DE DISCOS

ADMINISTRACIÓN DE DISCOS

Un **dispositivo de almacenamiento** (disco duro mecánico, SSD, microSD, pen drive, etc.) se organiza en una o más **particiones**, donde cada partición es una zona contigua de almacenamiento que tiene su propio sistema de archivos.

Todos los discos necesitan una **tabla de particiones** para indicar las particiones de un disco y sus límites. Existen múltiples formatos de tablas de particiones, como MBR (MS-DOS), GPT o BSD.

Cada partición necesita un **sistema de archivos** para poder almacenar datos, y para ello se le aplica un **formato** determinado ("formatear"), como ext4 (para ficheros en Linux), Swap, NTFS (para ficheros en Windows) o exFAT (para dispositivos extraíbles).

DISCOS

Los dispositivos SCSI (o discos) se referencian en Linux como archivos por bloques dentro del directorio `/dev`.

- Los discos duros y memorias flash en general se denominan mediante `sdx`, donde `x` es una letra minúscula que va de la `a` en adelante.

  El primer disco duro detectado se llamará `/dev/sda`, el segundo `/dev/sdb`, etc.

- Los CDs se denominan mediante `scdn` o `srn`, donde `n` es un número de `0` en adelante.

  El primer CD suele llamarse `/dev/sr0`.

PARTICIONES

Las particiones también se referencian en Linux como archivos por bloques en `/dev`.

- Se denominan añadiendo un número decimal al nombre del dispositivo (empezando por `1`).

  Por ejemplo, `/dev/sda1` y `/dev/sda2` representarían las primera y segunda particiones del primer disco duro del sistema.

Las particiones también pueden identificarse mediante su UUID (identificador único universal) o mediante su etiqueta.

FDISK

`fdisk` es una herramienta CLI que permite ver, crear y manipular tablas de particiones de un disco (creando, alterando o borrando particiones). Es compatible con múltiples formatos de tablas de partición, como GPT (UEFI), MBR (BIOS) o BSD.

Tiene dos formas de ejecutarse (ambas requieren permisos de administrador):

- `fdisk -l`: muestra la lista de dispositivos conectados y sus tablas de partición. Se le puede pasar un dispositivo como argumento, en cuyo caso mostrará su tabla de partición.
- `fdisk <dispositivo>`: muestra una sencilla consola de comandos de manipulación de tablas de partición para el dispositivo seleccionado. Las modificaciones no se aplicarán hasta ejecutar el comando `w` de aplicar los cambios y salir, o bien puede salirse sin aplicar los cambios con `q`.

MKFS

Tras crear particiones, hace falta **formatearlas** o **crear un sistema de archivos** en ellas para que puedan alojar ficheros. Para ello, en Linux existen comandos de la forma:

```
# mkfs.<formato> <partición>
```

Algunos formatos:

- `mkfs.ext4`: formatea una partición como ext4 (de Linux).
- `mkfs.fat`: formatea una partición como FAT32 (como CDs, tarjetas SD, pen drives, etc.).
- `mkfs.ntfs`: formatea una partición como NTFS (de Windows).

Ejemplo:

```
# mkfs.ext4 /dev/sdb2
```

MONTAR UN SISTEMA DE ARCHIVOS

En Linux, para usar un sistema de archivos externo al árbol de directorios raíz del sistema, se ha de **montar** primero **en un directorio** dentro de este:

```
# mount [-t <tipo>] [-o <opciones>] <partición> <ruta>
```

Se pueden especificar, opcionalmente, el formato (`<tipo>`) del sistema de ficheros y las `<opciones>` de montaje.

La **ruta (o punto) de montaje** es un directorio que debe existir previamente, y una vez montada la partición en ella, será la ruta de acceso al árbol raíz de la partición.

Para desmontar una partición, solo es necesario especificar el punto de montaje:

```
# umount <ruta>
```

MONTAR UN SISTEMA DE ARCHIVOS

Por ejemplo, si vemos (mediante `fdisk -l`) que una memoria USB conectada al equipo aparece como `/dev/sdb` y queremos montar la partición `/dev/sdb1`, hacemos:

```
# mkdir -p /mnt/usbdrive #Creamos el directorio si no existe
# mount /dev/sdb1 /mnt/usbdrive
```

Para desmontarla:

```
# umount /mnt/usbdrive
```

/ETC/FSTAB

Si queremos que los montajes se hagan automáticamente al iniciar el equipo, podemos configurarlos en el fichero de configuración `/etc/fstab`.

Este fichero contiene comentarios (comienzan por #) y una línea por cada partición que se desea montar, con los siguientes argumentos separados por espacios:

```
<file system>  <mount point>  <type>  <options>  <dump>  <pass>
```

- `<file system>`: nombre de la partición en `/dev`, etiqueta (`LABEL=<etiq>`) o UUID (`UUID=<uuid>`).
- `<mount point>`: punto de montaje.
- `<type>`: tipo (formato) del sistema de ficheros. Se puede poner `auto`.

/ETC/FSTAB

- `<options>`: opciones de montaje. Algunas conocidas:
  - `defaults`: asigna las opciones `rw`, `suid`, `dev`, `exec`, `auto`, `nouser` y `async`.
  - `auto`/`noauto`: será/no montado automáticamente durante el arranque (o con `mount -a`).
  - `exec`/`noexec`: permite/no permite la ejecución de binarios en el sistema de ficheros.
  - `ro`/`rw`: monta el sistema de archivos como solo lectura/lectura y escritura.
  - `user`/`nouser`: permite que cualquier usuario monte el sistema de archivos.
- `<dump>`: utilizado por el programa `dump` (si está instalado) para determinar si hará una copia de seguridad de la partición (valor `1`) o la ignorará (valor `0`).
- `<pass>`: orden en el que los sistemas de archivos serán comprobados. Tiene un valor de `0` (no será comprobado), `1` o `2`.

/ETC/FSTAB

Para montar con `mount` un sistema de ficheros definido en `/etc/fstab`, se debe especificar solamente el nombre de la partición o el punto de montaje, pero no ambos:

```
# mount <ruta>
```

Para montar todos los sistemas de ficheros que aparecen en `/etc/fstab` con la opción `auto` que falten por montar:

```
# mount -a
```

LSBLK

Lista la información de todos los dispositivos por bloques disponibles del sistema (discos y particiones):

```
$ lsblk [<dispositivos...>]
```

Muestra la lista de particiones de un disco junto a dicho disco en formato árbol, junto a datos como tamaño, si es de solo lectura (`RO`) y puntos de montaje (si está montado).

Con la opción `-f`, muestra información sobre los sistemas de archivos, como formato, UUID y espacio disponible.

Si se le pone argumentos, muestra solo los datos de esos dispositivos.

DF

Muestra el uso de disco de los sistemas de ficheros montados en el sistema:

```
$ df [<ficheros...>]
```

Si se incluyen rutas como argumentos, mostrará el uso de disco de los sistemas de ficheros que contengan a dichos ficheros o directorios.

Opciones:

- `-a` incluye todos los sistemas de ficheros.
- `-h` para mostrar los tamaños en potencias de 1024 (`1023M`).
- `-H` para mostrar los tamaños en potencias de 1000 (`1.1G`).

GESTIÓN DE PAQUETES EN LA FAMILIA DEBIAN

COMANDOS APT, DPKG-RECONFIGURE

ADMINISTRACIÓN DE PAQUETES EN DEBIAN

ADMINISTRACIÓN DE PAQUETES

Un **paquete** en Linux es una pieza de software empaquetada para instalarse en un sistema. Un paquete puede incluir aplicaciones, bibliotecas, drivers, archivos de configuración, etc.

Las distribuciones basadas en Debian usan el formato `.deb` para paquetes que contienen su código y una lista de dependencias, que son paquetes adicionales que será necesario instalar para su correcto funcionamiento.

Desde la versión 16.04, Ubuntu creó su propio formato de paquetes `snap`, que contienen sus propias dependencias, por lo que pesan más pero se ahorran algunos problemas al no necesitar otros paquetes para funcionar.

ADMINISTRACIÓN DE PAQUETES

En la familia Debian (Debian y derivados), la administración de paquetes se realiza mediante:

- `dpkg`: gestor de paquetes `deb`. No se suele usar directamente, sino que otras interfaces de alto nivel lo usan internamente para las operaciones sobre paquetes.
- `apt`: interfaz de gestión de paquetes amigable con el usuario. Tiene las funciones básicas para la administración de paquetes. Es la que más se usa. Es la evolución de `apt-get`.
- `apt-cache`: comando para consultar la caché de `apt`.
- `snap`: gestor de paquetes `snap` (Ubuntu).

ORÍGENES DEL SOFTWARE

Para que `apt` funcione es necesaria una correcta configuración de los **orígenes del software o repositorios**, que son los lugares desde donde el sistema solicita los paquetes, ya sea para instalarlos o para actualizarlos.

Los repositorios contienen una colección de paquetes organizados en diferentes categorías como `main`, `contrib` o `non-free`. Es importante usar siempre orígenes confiables.

El archivo que almacena la configuración de los orígenes está en `/etc/apt/sources.list`.

LISTA DE PAQUETES

Es necesario que `apt` **actualice su lista de paquetes** y versiones disponibles en los repositorios con el siguiente comando:

```
$ apt update
```

Siempre que se vaya a actualizar o instalar paquetes, será necesario ejecutar este comando primero, o no se instalarían las versiones más recientes de dichos paquetes.

ACTUALIZACIONES

En general, existen dos tipos de actualizaciones:

- **Actualizaciones de seguridad**: corrigen fallos y agujeros de seguridad.
- **Actualizaciones de versiones**: añaden nuevas características. Son menos frecuentes que las de seguridad.

Normalmente, las actualizaciones en distribuciones Linux son rápidas y estables, pero siempre es recomendable hacer una copia de seguridad de los datos más importantes por si algo sale mal.

ACTUALIZACIONES

Para realizar las **actualizaciones**, ya sean del núcleo de Linux o de librerías y programas, tenemos dos posibles comandos:

```
$ apt upgrade
$ apt full-upgrade
```

El primero actualiza los paquetes a la última versión disponible. Si algún paquete tiene nuevas dependencias, se instalarán también. No desinstala las dependencias que dejaron de usarse.

El segundo (también denominado `dist-upgrade`) hace lo anterior y, además, en caso de necesitar eliminar paquetes por cambios en las dependencias de algún paquete, los elimina.

OTROS COMANDOS DE GESTIÓN DE PAQUETES

- `apt search <regexp>`: busca entre la lista de paquetes disponibles.
- `apt show <paquete>`: muestra información de un paquete, incluyendo nombre, versión más reciente, tamaño de la descarga, dependencias, etc.
- `apt install <paquetes...>`: instala uno o varios paquetes. Si dichos paquetes tienen dependencias no instaladas, estas se instalarán también.
- `apt remove <paquetes...>`: desinstala uno o varios paquetes. Elimina los ficheros de código, pero puede dejar archivos de configuración sin borrar.
- `apt purge <paquetes...>`: desinstala uno o varios paquetes, y además elimina los archivos de configuración.

OTROS COMANDOS DE GESTIÓN DE PAQUETES

- `apt autoremove`: desinstala los paquetes instalados automáticamente como dependencias y que ya no están en uso.
- `dpkg-reconfigure <paquete>`: muestra el diálogo de configuración *wizard* que se mostró al ser instalado el paquete (si lo hubo).

LOGS, PROTOCOLO SSH.

COMANDOS IP, NSLOOKUP, PING, NETCAT, SS, CURL, WGET, SSH, SCP Y SFTP Y LOGGER

COMANDOS DE RED

IP

Muestra o manipula configuraciones de enrutamiento, interfaces y dispositivos de red.

Para ver las interfaces de red a nivel de enlace:

```
$ ip link show [<int>]
```

Para ver las interfaces y su dirección IP:

```
$ ip address show [<int>]
```

Para ver la tabla de enrutamiento:

```
$ ip route list [dev <int>]
```

Para ver la tabla de vecinos de la red:

```
$ ip neigh show [dev <int>]
```

Los comandos anteriores permiten especificar la interfaz que mostrar (`<int>`).

Se pueden usar abreviaciones para estos comandos (`ip l`, `ip a`, `ip r`, `ip n`).

Con `-4`, muestra direcciones IPv4.

Con `-br`, muestra una salida breve.

Con `-j`, muestra la salida en formato JSON.

PING

Manda paquetes ICMP **Echo Request** a otros equipos de la red y espera su respuesta. Útil para comprobar la comunicación entre equipos o redes:

```
$ ping <host>
```

Donde `<host>` es la dirección IP o nombre de dominio del host con el que se quiere comunicar.

Cualquier equipo o router conectado a una red y escuchando será capaz de responder a un ping. Las respuestas son paquetes ICMP **Echo Reply**, aunque también se pueden recibir otros paquetes ICMP como **Redirect** o **Destination Unreachable**.

TRACEROUTE

El siguiente comando ayuda a diagnosticar los problemas de una red (tanto paquetes que no llegan como latencias o cuellos de botella en una red):

```
$ traceroute <IP_o_nombre>
```

Este comando registra la ruta por la que tiene que pasar un paquete, mostrando los nodos por los que tiene que pasar hasta llegar al destino.

Alternativas:

- `tracepath`, parecido a `traceroute`, pero más sencillo.
- `mtr` combina funcionalidades de `ping` y `traceroute` en una sola herramienta:

```
$ mtr <IP_o_nombre>
```

DNS

Para ver los servidores DNS configurados en el sistema:

- En sistemas con `systemd-resolved`:

```
$ resolvectl
```

- En sistemas sin `systemd-resolved`:

```
$ cat resolv.conf
```

Para resolver nombres de dominio (o hacer resoluciones inversas) en la terminal:

```
$ nslookup <nombre_o_IP> [<servidor_DNS>]
```

`nslookup` tiene también un modo interactivo.

Otros comandos de resolución de nombres más avanzados son `dig` y `drill`.

NETCAT

El comando `nc` o `netcat` permite configurar rápidamente un cliente y/o un servidor para probar una comunicación o enviar datos.

Para establecer un cliente que se conecte a un servidor (`<host_dest>`) a través de un puerto (`<puerto_dest>`):

```
$ nc [-v] [-u] <host_dest> <puerto_dest>
```

Para establecer un servidor que escuche por un puerto (`<puerto_orig>`):

```
$ nc [-v] [-u] -lp <puerto_orig>
```

Una vez establecida una comunicación entre cliente y servidor, se podrán intercambiar mensajes entre ellos mediante la entrada y salida estándar de ambos comandos.

NETCAT

Un cliente (o servidor) también puede leer la entrada estándar de otra fuente:

```
$ cat req.txt | nc -C google.es 80
```

Opciones de `nc`:

- `-v`: verbose (indica por stderr si se ha establecido una conexión, si se ha cerrado, ...)
- `-u`: usa UDP. Sin esta opción, usa TCP.
- `-l`: de *listen* (para servidor). Escucha por el puerto especificado por `-p`.
- `-s <IP>`: especifica la dirección IP de origen.
- `-p <puerto>`: especifica el puerto de origen (obligatorio si se usa `-l`).
- `-C`: envía los saltos de línea (LF, estilo UNIX) como CRLF (estilo Windows).

SS

`ss` es una utilidad para ver las conexiones activas y sockets del sistema.

Algunas opciones útiles son:

- `-a`: para ver conexiones activas, sockets y puertos de escucha.
- `-l`: para ver solo puertos de escucha.
- `-t`/`-u`: para ver solo sockets TCP/UDP.
- `-p`: para ver el proceso que usa el puerto.
- `-e`: para ver información extendida.
- `-n`: no intenta resolver los nombres de IP y puertos, sino que los muestra numéricos.

Para ver puertos TCP de escucha:

```
$ ss -ltn
```

Toda la información de puertos de escucha:

```
$ ss -tunelp
```

Para ver los sockets con puerto de origen 22:

```
$ ss -an sport = 22
```

`ss` es una mejor alternativa a `netstat`, aunque comparte muchas de sus opciones.

CURL

Transfiere datos desde o hacia un servidor. Su uso más básico es el de hacer una petición HTTP a un servidor:

```
$ curl <URL>
```

En la URL se puede incluir o no el protocolo (si no, se asume HTTP). El resto de su sintaxis depende del protocolo.

`curl` es compatible con muchos protocolos y sus versiones seguras, como HTTP, FTP, IMAP, POP3, SMTP, SMB, LDAP, RTMP, etc. También soporta TELNET, SCP y SFTP.

Por defecto, se limita a mostrar el cuerpo de la respuesta en la salida estándar.

`curl` tiene muchas opciones para personalizar el mensaje de petición.

CURL - USOS ÚTILES

Algunos usos útiles de CURL:

- Descargar y ejecutar un script de instalación (solo fuentes confiables + HTTPS):

```
$ curl -fsSL https://get.docker.com | sudo sh
```

- Extraer datos (JSON) de una API:

```
$ curl -LsS dummyjson.com/test | jq '.method'
```

- Ver tu dirección IP pública:

```
$ curl ifconfig.co
```

- Ver el tiempo:

```
$ curl wttr.in
```

CURL - OPCIONES

Algunas opciones:

- `-L`: sigue redirecciones automáticamente.
- `-I`/`--head`: manda una petición HEAD para ver solo las cabeceras de la respuesta.
- `-i`/`--include`: incluye las cabeceras de la respuesta en la salida.
- `-H "<cabecera>: <valor>"`: define una cabecera personalizada.
- `-X <método>`: define el método HTTP a usar en la petición.
- `-F <clave>=<valor>`: añade un campo de formulario en una petición POST.
- `--json <JSON>`: pone datos en formato JSON en una petición POST.

CURL - OPCIONES

- `-o <fichero>`/`-O`: guarda la salida en un fichero en vez de la salida estándar.
- `--remove-on-error`: borra el fichero (`-o`/`-O`) si ocurre un error en la transmisión.
- `--proto '=https' --tlsv1.2`: fuerza el uso de HTTPS con TLS 1.2 o posterior.
- `-sS`: no muestra la barra de progreso de una transmisión.
- `-v`/`--verbose`: muestra mensajes y las cabeceras de petición y de respuesta.
- `--trace <fichero>`: escribe la traza completa de los datos entrantes y salientes.
- `-f`/`--fail`: si falla, no muestra el cuerpo de la respuesta y sale con error 22.
- `-m <segundos>`: tiempo máximo que puede durar el intento de conexión.

WGET

Descarga archivo(s) de un servidor remoto de forma no interactiva:

```
$ wget <URL>
```

Con las siguientes opciones, **`wget` puede descargar páginas web enteras**:

- `-r/--recursive`: descarga también las páginas referenciadas en los enlaces de la página web (`<URL>`) con hasta 5 grados de profundidad (personalizable con `-l`).
- `-l <num>`: profundidad máxima al descargar sitios web con `-r`. `0` es infinita.
- `-p`: descarga también los recursos de la página (CSS, JS, imágenes, etc.)
- `-k`: convierte los enlaces para que se pueda visualizar localmente de forma correcta.
- `-H`: permite descargar de otros hosts. Sin esta opción, solo descarga del mismo sitio web.

SSH

Este comando usa el **protocolo SSH** (*Secure SHell*) para iniciar sesión en un **shell remoto** de forma segura (mediante un túnel cifrado SSH). Uso básico:

```
$ ssh <usuario>@<host>
```

Donde `<host>` es la dirección IP o nombre del equipo remoto y `<usuario>` es el nombre de usuario remoto con el que se quiere iniciar sesión.

Para que funcione, el equipo cliente debe tener instalado un cliente SSH (`openssh-client`) y el equipo remoto, un servidor SSH (`openssh-server`).

Si puede realizar la conexión SSH, pedirá la contraseña de `<usuario>` (a menos que se haya configurado otro método de autenticación) y, si es correcta, mostrará un shell.

SSH - REENVIAR PUERTOS

`ssh` ofrece varias opciones para **reenviar puertos** entre cliente y servidor:

- `-L <puerto_local>:<host_remoto>:<puerto_remoto>`: reenvía las conexiones locales a `localhost:<puerto_local>` hacia la conexión remota `<host_remoto>:<puerto_remoto>` mediante un túnel cifrado SSH.
- `-R <puerto_remoto>:<host_local>:<puerto_local>`: hace justo lo opuesto a `-L`: reenvía conexiones remotas (`localhost:<puerto_remoto>`) hacia una conexión local (`<host_local>:<puerto_local>`).

Ej: para acceder localmente (`localhost:8080`) a un puerto remoto (`127.0.0.1:80`):

```
$ ssh -N -L 8080:127.0.0.1:80 user@192.168.1.5
```

SSH - OPCIONES

Algunas opciones útiles de SSH son:

- `-N`: no abre un shell remoto. Es lo apropiado cuando se reenvían puertos (`-L`, `-R`).
- `-p <puerto>`: define el puerto TCP con el que conectarse por SSH al host remoto (debe coincidir con el puerto de escucha configurado en el servidor SSH).

  Por defecto, SSH usa el puerto estándar SSH (TCP 22), que es el puerto que recibe más ataques de fuerza bruta, por lo que no se recomienda exponer este puerto a internet.
- `-X`: habilita *X11 forwarding*, lo que permite abrir aplicaciones gráficas de forma remota (solo si el cliente tiene un cliente de ventanas X11).

SSH - AUTENTICACIÓN

Para establecer una conexión SSH se necesita una prueba de autenticación del cliente, normalmente mediante la contraseña, aunque también se puede realizar mediante **claves SSH**. Para generar un par de claves SSH (privada y pública):

```
$ ssh-keygen [-t <alg>]
```

La opción `-t`, para seleccionar el **algoritmo** de clave pública, es opcional. Actualmente se recomienda el uso del algoritmo `ed25519`.

Este comando pide introducir una **passphrase** para cifrar la clave privada (mayor seguridad), aunque es opcional.

Al terminar genera dos ficheros en `~/.ssh/`: la clave privada (`id_<alg>`) y la pública (`id_<alg>.pub`)

SSH - AUTENTICACIÓN

Para usar estas claves del cliente como método de autenticación para conectarte como `<usuario>`, copia la clave pública del cliente (`~/.ssh/id_<alg>.pub`) en una nueva línea de `~/.ssh/authorized_keys` en el directorio personal de `<usuario>` en el servidor, o mediante el siguiente comando desde el cliente:

```
$ ssh-copy-id <usuario>@<host>
```

De esta forma, al volver a conectar por SSH a *`<usuario>`*`@`*`<host>`* desde este cliente, no volverá a pedir **contraseña** (aunque sí la **passphrase** de la clave SSH, si se le definió una).

SSH - AUTENTICACIÓN DEL SERVIDOR

La primera vez que un cliente SSH se conecta a un servidor SSH que no está autenticado (por certificado firmado por una CA de confianza), preguntará al usuario si confía en él y, si responde que sí, lo añadirá a la lista de equipos conocidos del cliente (`~/.ssh/known_hosts`).

Este archivo, en la carpeta personal del usuario que realizó la conexión, guarda una línea por cada nombre o IP que usó para conectar con el servidor y por cada huella digital del servidor.

Cada servidor SSH tiene una huella única, por lo que al iniciar una conexión SSH, el cliente busca en `known_hosts` el host al que se está conectando y, si encuentra alguna línea, comparará las huellas.

Si la huella de un servidor no coincide con la huella de un servidor al que se conectó usando el mismo nombre o IP, el cliente bloqueará la conexión, ya que podría tratarse de una **suplantación del servidor**.

Si el usuario está seguro de que no es una suplantación, tendrá que borrar la línea o líneas que correspondan con ese nombre o IP del archivo `known_hosts` y volver a intentar la conexión.

SCP

Establece un **túnel SSH** para transferir archivos entre diferentes equipos (siempre que el equipo remoto sea un servidor SSH).

Para transferir un archivo/carpeta local a un equipo remoto:

```
$ scp [-r] <ruta_origen> <usuario>@<host>:<ruta_destino>
```

Para transferir un archivo/carpeta remota al equipo local:

```
$ scp [-r] <usuario>@<host>:<ruta_origen> <ruta_destino>
```

Su uso es similar al comando `cp`. Si lo que se quiere transferir es o incluye un directorio, será necesario el uso de la opción `-r` (recursivo).

SFTP

Establece un **túnel SSH** para establecer una sesión SFTP (FTP sobre SSH) con un equipo remoto (siempre que este sea un servidor SSH) para tranferir ficheros. Uso:

```
$ sftp <usuario>@<host>
```

La sintaxis básica es idéntica a la del comando `ssh`, y además pedirá contraseña (si permite el inicio de sesión con contraseña).

Al conectarse con éxito, se abrirá una consola con comandos SFTP. Algunos comandos SFTP son: `help` (muestra una lista con los comandos disponibles), `get` (descargar), `put` (subir), `ls` (`ls` remoto), `lls` (`ls` local), `cd` (`cd` remoto) y `lcd` (`cd` local).

OTROS COMANDOS DE RED

- `netstat`: para ver información de conexiones y puertos abiertos (similar a `ss`).
- `iftop`: para monitorizar el uso de la red. Muestra conexiones de red y anchos de banda.
- `tcpdump`: captura en tiempo real información de paquetes de red (sniffer).
- `tshark`: captura y filtra paquetes de red en tiempo real (sniffer). Es similar a Wireshark, pero para uso en terminal.
- `whois <dominio>`: consulta información sobre un nombre de dominio (de resolución directa o inversa), como quién lo posee, fecha de registro y/o de expiración.

LOGS

Los **logs** recolectan mensajes de eventos de la actividad del sistema, como el arranque, intentos de inicio de sesión o mensajes y errores de diferentes servicios del sistema.

En Linux, los logs se guardan por defecto en ficheros dentro de `/var/log` de cada equipo, aunque se pueden configurar otras rutas o incluso configurar equipos para que envíen sus logs a un servidor log central.

Los logs se pueden verse abriendo los archivos que los guardan. Por ejemplo:

```
$ tail -f /var/log/syslog
```

O con comandos como `dmesg` (para logs del kernel) o `journalctl` (systemd):

```
$ journalctl -xe
```

ORIGEN DE LOS MENSAJES LOG

Clasificación (subsistemas) de los logs	
`auth`, `authpriv`	Autenticación
`cron`	Tareas programadas con `cron` o `at`
`daemon`	Demonios sin clasificación (DNS, NTP)
`kern`	Del núcleo (*kernel*) del SO.
`lpr`	Impresión.
`mail`	Correo electrónico.
`syslog`	Servidor `syslogd`
`user`	De usuario, genérico
`local0`, ..., `local7`	Reservado para uso local
`ftp`, `news`, `uucp`	Otros

PRIORIDAD DE LOS MENSAJES LOG

Código	Keyword	Severidad
0	`emerg`	El sistema no se puede utilizar
1	`alert`	Se necesita una acción inmediata
2	`crit`	Condiciones del sistema críticas
3	`err`	Un error, que puede no ser limitante
4	`warning`	Advertencia, potencial error
5	`notice`	Anuncio importante en condiciones normales
6	`info`	Informativo
7	`debug`	Mensaje para pruebas del desarrollador

LOGGER

Envía mensajes al **log** del sistema:

```
$ logger [<mensaje>]
```

Si se especifica `<mensaje>`, envía ese mensaje. Si no, envía lo que le llega por la entrada estándar. Algunas opciones:

- `-p <prioridad>`: establece la prioridad (o el par origen.prioridad) del mensaje. Por defecto es `user.notice`.
- `-n <host>`: manda el log a un servidor remoto.
- `-P <puerto>`: usa un puerto determinado.
- `--tcp`/`--udp`: usa el protocolo de transporte TCP/UDP.

COMANDOS CLEAR, DIFF, DATE, SLEEP, BC, XARGS, EXIT, LOGOUT, SHUTDOWN, POWEROFF, REBOOT

OTROS COMANDOS

CLEAR

Borra toda la pantalla de comandos:

```
$ clear
```

Equivale al atajo `Ctrl+L`.

Con la opción `-x`, conserva el buffer anterior (sigue pudiéndose hacer scroll hacia arriba).

DIFF

Encuentra las diferencias, por líneas, entre ficheros, carpetas o ambos:

```
$ diff <archivos...>
```

Con la opción `-y`, muestra las diferencias de dos ficheros o carpetas en dos columnas (más visual):

```
$ diff -y <archivo1> <archivo2>
```

Con la opción `-r`, compara recursivamente subdirectorios.

DATE

Muestra la fecha y hora actual:

```
$ date [+<formato>]
```

Si se incluye la opción `-d <fecha>`, usa `<fecha>` en lugar de la fecha actual. Puede ser absoluta (`2020-12-31`) o relativa (`+8days`, `-5mins`, `+1year`, etc.).

Por defecto, muestra la fecha en el formato configurado en el sistema (`locale`).

Se le puede especificar un formato personalizado (`+<formato>`) a la hora de imprimirlo. Por ejemplo:

```
$ date -d 'sep 12 2007 10:30pm' '+%Y-%m-%d %H:%M:%S'
2007-09-12 22:30:00
```

SLEEP

Hace una pausa de un tiempo determinado:

```
$ sleep <tiempo>
```

`<tiempo>` es un número (entero o decimal) de segundos, salvo si se le añade un sufijo:

- `s` para segundos (predeterminado).
- `m` para minutos.
- `h` para horas.
- `d` para días.

BC

Calculadora básica. Puede aceptar expresiones aritméticas por la entrada estándar o ejecutarse de forma interactiva (sin redirecciones ni tuberías):

```
$ bc
```

Admite los operadores `+`, `-`, `*`, `/`, `%` (módulo), `^` (potencia), `=` (asignación), `()` (precedencia), comparadores (`<`, `<=`, `>`, `>=`, `==`, `!=`), `!` (no), `&&` (y), `||` (o) y la sentencia `quit` (salir), etc.

Con la opción `-l`, incluye funciones de la biblioteca matemática estándar:

- `s(x)`, `c(x)`, `a(x)`: seno, coseno y arco tangente de `x`.
- `l(x)`, `e(x)`: logaritmo natural ($\ln(x)$) y exponencial (e^x) de `x`.

XARGS

Para pasar los resultados de un comando como argumentos de otro comando:

```
$ <cmd1> | xargs <cmd2> [<args...>]
```

`xargs` lee elementos de la entrada estándar, separados por espacios (o tabuladores o saltos de línea), los pone en `<cmd2>` como argumentos (después de `<args...>`) y lo ejecuta.

- Con `-d <delim>`, se personaliza el delimitador entre elementos (o `-0` para carácter nulo).
- Con `-I <str>`, reemplaza las ocurrencias de `<str>` en `<args...>` por cada elemento.
- Con `-n <n>`, limita el número de argumentos por comando (ejecutando varios comandos).
- Con `-P <n>`, ejecuta hasta `<n>` procesos simultáneos (para usar junto a `-n`).

Por ejemplo:

```
$ cut -d: -f1 < /etc/passwd | sort | xargs -I{} echo \"{}\"
$ find /tmp -name core -type f -print0 | xargs -0 /bin/rm -f
```

EXIT

Para salir de un (sub)shell interactivo (como `ssh` o `su`), se puede usar cualquiera de los siguientes comandos:

```
$ exit [<código_error>]
$ logout [<código_error>]
```

`logout` solo se puede usar en shells con inicio de sesión.

`exit` se puede usar en cualquier shell, también en shells no interactivos (scripts).

Ambos comandos pueden recibir un argumento opcional: el **código de error**.

- Si se omite, se usará el código de error del último comando ejecutado.

APAGAR EL EQUIPO

Nota: los comandos presentados a continuación requieren permisos de administrador.

Para **apagar el equipo**, ejecuta uno de los siguientes comandos:

```
# shutdown now
# poweroff
```

Si no se especifica el argumento `now` (o un número de minutos `+n`, o una hora concreta `HH:MM`), el sistema esperará 1 minuto antes de apagarse.

Para **reiniciar el equipo**, ejecuta uno de los siguientes comandos:

```
# shutdown -r now
# reboot
```

ALGUNAS EXPANSIONES ÚTILES DEL SHELL

EXPANSIONES

EXPANSIONES

Una **expansión** del shell es una construcción sintáctica que **se expande a un cierto valor justo antes de ser ejecutada**, por lo que el comando verá el valor resultante.

Ejemplo:

```
$ cp *.txt copia_logs
```

Antes de ejecutarse, `*.txt` se expandirá (según el contenido del directorio actual) y el comando resultante será algo como:

```
$ cp fich1.txt fich2.txt instrucciones.txt copia_logs
```

Si no se quiere que se lleve a cabo una expansión (es decir, tratarla como texto literal), se pueden usar caracteres de escape o entrecomillados.

EXPANSIONES

Los siguientes caracteres, si se escriben sin entrecomillar, son expansiones:

- La virgulilla (**~**) es equivalente a `$HOME`, el directorio personal del usuario actual.
- El asterisco (*****) es equivalente a uno o más caracteres en el nombre de un archivo.

  Ej: `echo *.c` lista todos los archivos con extensión `.c` del directorio actual.

Solo para `bash`:

- El signo de interrogación (`?`) es equivalente a un único carácter.

  Ej: `echo curso.te?` lista los archivos `curso.tea`, `curso.teb`, etc. (solo los que existan).
- Un conjunto de caracteres entre corchetes es equivalente a cualquier carácter del conjunto.

  Ej: `echo curso.t[ao]x` lista `curso.tax` y `curso.tox` (solo los que existan).

EXPANSIONES

Las siguientes expansiones (solo `bash`) permiten generar listas de elementos:

- `abc{3..8}` o `abc{3..8..1}`

  Equivalente a: `abc3 abc4 abc5 abc6 abc7 abc8`

  Se sustituye por una lista de tantos elementos como haya dentro del rango (ambos incluidos), concatenados con el prefijo/sufijo que le acompañe. Se puede especificar el paso que se sumará.

- `abc{def,01,}`

  Equivalente a: `abcdef abc01 abc`

  Se sustituye por una lista de tantos elementos como haya dentro de las llaves, concatenados con el prefijo/sufijo que le acompañe.

EXPANSIONES

Las siguientes expansiones con `$` se pueden usar tanto fuera como dentro de un string entrecomillado (`""`):

- `$<variable>`

  Se sustituye por el valor de la variable.

- `${<variable>}`

  Se sustituye por el valor de la variable. Con esta sintaxis, se puede manipular el resultado.

- `$(<comandos...>)` o `` `<comandos...>` ``

  Se sustituye por la salida estándar del grupo de comandos.

- `$((<expresión>))`

  Se sustituye por el resultado de calcular la expresión aritmética.

${...}

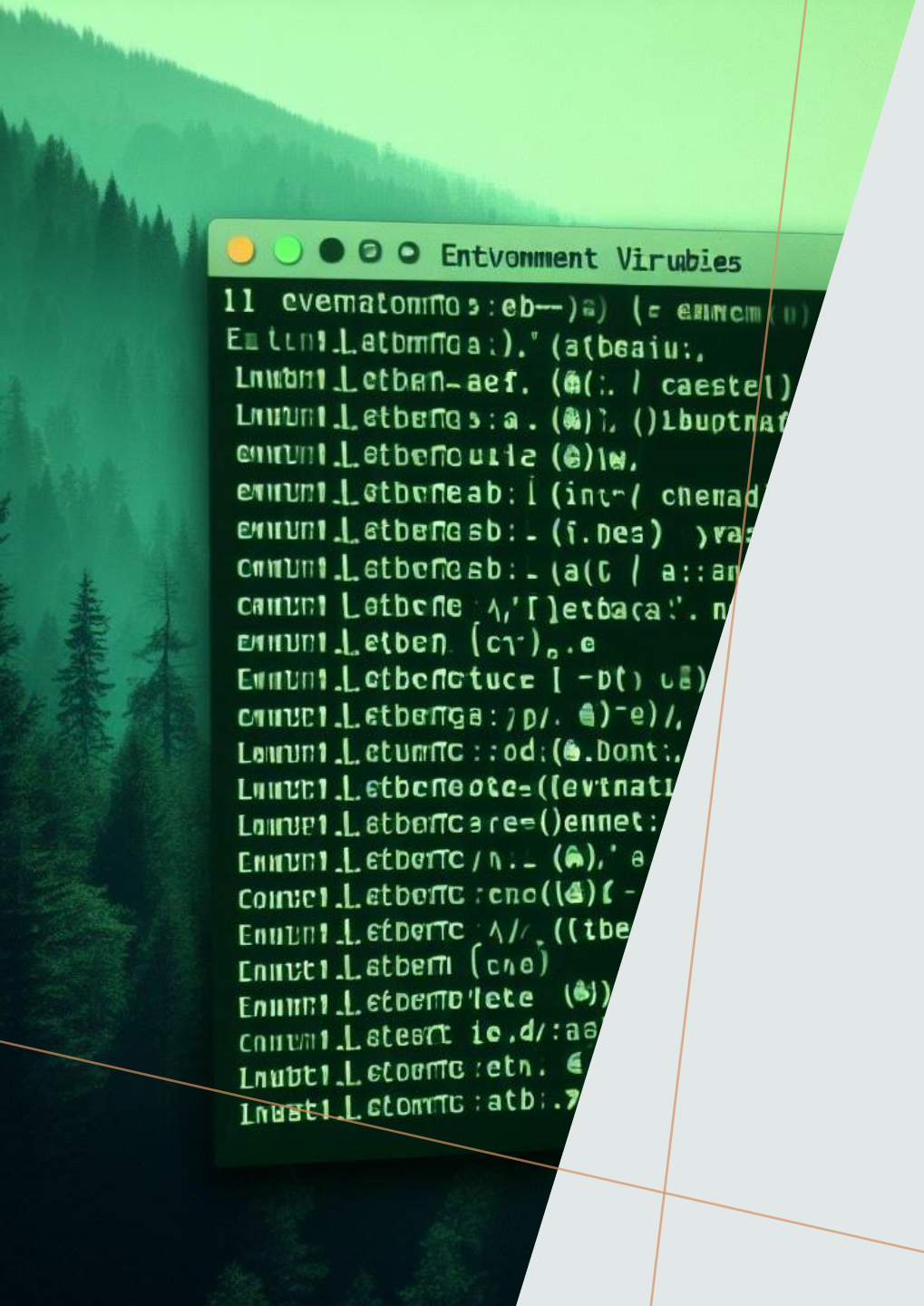
Las expansiones ${ } permiten algunas operaciones rápidas:

- ${<var>:-<valor>} #Devuelve el valor de la variable o, si está vacía, devuelve un valor por defecto
- ${<var>:=<valor>} #Si está vacía, le da valor a la variable
- ${<var>:?<errormsg>} #Si <var> está vacía devuelve mensaje de error y sale
- ${#<var>} #Devuelve la longitud de la variable
- ${<var>#<patrón>} #Elimina el patrón inicial de la variable (## para llegar al final)
- ${<var>%<patrón>} #Elimina el patrón final de la variable (%% para llegar al principio)

${...}

Las siguientes expansiones solo las admite `bash`:

- `${<var>/<patrón>/<reemplazo>}` `#Buscar un patrón y reemplazarlo`
- `${<var>//<patrón>/<reemplazo>}` `#Reemplazar todas las ocurrencias`
- `${<var>:<inicio>:<longitud>}` `#Recortar un string (si no se especifica longitud, hasta el final)`
- `${<var>^^}` `#Convertir a mayúsculas (un solo ^ para letra capital)`
- `${<var>,,}` `#Convertir a minúsculas`



MANIPULACIÓN DE VARIABLES, EJEMPLOS, RESULTADO DE UN COMANDO, ALIAS

COMANDOS PRINTENV, SET, UNSET, ALIAS, EVAL

VARIABLES Y ALIAS

VARIABLES

El **entorno** es un área que el shell crea cada vez que inicia una sesión que contiene variables que definen las propiedades del sistema y variables creadas por el usuario.

- Las **variables de entorno** son variables definidas para el shell actual y heredadas por cualquier shell o proceso secundario de este shell. Para ver las variables de entorno actuales, ejecuta el comando `printenv` (o `env`, que también puede ejecutar un programa modificando sus variables de entorno).
- Las **variables de shell** son variables que se encuentran exclusivamente dentro del shell en el que se crearon.

Los **nombres de variables** solo pueden contener **letras**, **números** y **_** (underscore), y no deben empezar por un número.

Para ver todas las variables y funciones definidas en la shell actual, ejecuta el built-in `set`.

VARIABLES - MANIPULACIÓN

Para crear (o cambiar el valor a) una **variable de shell**:

```
$ <nombre>=<valor>
```

Para crear (o cambiar el valor a) una **variable de entorno**:

```
$ export <nombre>=<valor>
```

Para ejecutar un comando dando valor a una variable del entorno del comando (el shell actual no verá esta variable):

```
$ <nombre>=<valor> <comando>
```

Para **expandir** (leer) **una variable**:

```
$ $<variable>
```

Para convertir una variable de shell ya creada en una de entorno:

```
$ export <nombre>
```

Para convertir una variable de entorno ya creada en una de shell:

```
$ export -n <nombre>
```

Para eliminar una variable:

```
$ unset <nombre>
```

VARIABLES PREDEFINIDAS

- **`SHELL`**: intérprete del shell actual.
- **`TERM`**: emulador de terminal actual, si lo hay. Indica si es un shell interactivo o no.
- **`USER`**: usuario del shell actual.
- **`PWD`**: directorio actual de trabajo (*Print Working Directory*, pwd).
- **`OLDPWD`**: directorio anterior de trabajo (se usa para volver con `cd -`).
- **`PATH`**: lista de directorios que el sistema comprobará cuando busque comandos (cuando se escriben con rutas relativas). Por ejemplo, el comando `cat` se buscará en las rutas de `PATH`, se encontrará en `/usr/bin` y se ejecutará `/usr/bin/cat`.
- **`HOME`**: (sinónimo de `~`) directorio personal del usuario actual. Cada usuario tiene un directorio personal (o no). El directorio personal del usuario `root` es `/root`.
- **`HOSTNAME`**, **`PS1`**, **`LINES`**, **`COLUMNS`**, ...

RESULTADO DE UN COMANDO

Los comandos del shell, y los programas en general, devuelven un **código de error** numérico al terminar la ejecución. Ese código se almacena en la variable especial `?`.

Para ver la terminación de un programa:

```
$ cat noexis
cat: noexis: No such file or directory
$ echo $?
1
```

El valor `0` indica que el programa se ejecutó correctamente. Cualquier otro valor (positivo o negativo) implica que hubo algún problema.

TIPOS DE DATOS

En `bash` no hay tipos de datos, sino que toda variable se guarda como cadenas de texto, pudiendo contener números, caracteres individuales, textos, fechas, etc.

ALIAS

Se puede crear un alias de cualquier comando con o sin argumentos con la siguiente sintaxis:

```
$ alias <nombre>=<comando>
```

Es frecuente poner alias en el archivo `.bashrc` para que se apliquen automáticamente al abrir una shell y personalizar la experiencia.

Ejemplos:

```
$ alias ls='ls -l'
$ ls /etc/interfaces        #Equivale a ls -l /etc/interfaces
$ alias ls='sudo rm -rf /' #Cuidado
```

EVAL

Evalúa argumentos como un comando:

```
$ eval <argumentos...>
```

Concatena los argumentos (con un espacio entre medias) en una sola cadena de texto y la ejecuta como un comando de shell.

Ejemplo:

```
$ a=15 b=a
$ c='$'$b
$ echo $c
$b
$ eval echo $c
15
```

CONDICIONES, OPERADORES LÓGICOS, COMMANDO SEC

COMANDOS TEST, [, SEQ Y SENTENCIAS IF, CASE, WHILE, FOR

SENTENCIAS DE CONTROL Y BUCLES

IF

Ejecuta unos comandos si y solo si se cumple una condición. Sintaxis:

```
if <condición>; then
    <comandos...> # Solo si la condición se cumple
fi
```

Aunque en la `<condición>` es frecuente emplear la sentencia `[[` (bash) o `[`, se puede usar cualquier comando, donde un resultado (`$?`) de `0` se interpretará como **verdadero** y un resultado distinto de `0` se evaluará como **falso**.

IF..ELSE

Como la sentencia `if`, pero ejecutando comandos alternativos en caso de que la condición no se cumpla. Sintaxis:

```
if <condición>; then
    <comandos...> # Cuando la condición se cumple
else
    <comandos...> # Cuando la condición no se cumple
fi
```

IF..ELIF..ELSE

Se pueden agregar tantas alternativas con condición `elif` como se quieran. Al ejecutarse, se compararán condiciones y entrará en la primera que resulte correcta. Sintaxis:

```
if <condición>; then
    <comandos...> # Cuando la primera condición se cumple
elif <condición>; then
    <comandos...> # Cuando la primera no se ha cumplido pero esta sí
elif <condición>; then
    <comandos...> # Cuando las anteriores no se han cumplido pero esta sí
# ...
else
    <comandos...> # Cuando ninguna condición se ha cumplido
fi
```

CASE

Si se quiere ejecutar comandos según el valor de una expresión (normalmente, una variable), la sentencia `case` es una alternativa a `if..elif..else`.

En `case`, se ejecutará el bloque de comandos que corresponda con el valor de `<expr>`.

El valor especial `*`, si se incluye, representa cualquier valor que no corresponda con ninguno de los valores anteriores.

```
case <expr> in
  <valor1>)
    <comandos si expr=valor1...>
    ;;
  <valor2>)
    <comandos si expr=valor2...>
    ;;
  # ...
  *)
    <comandos si ninguno...>
    ;;
esac
```

CONDICIONES

Una condición se puede escribir de las siguientes maneras:

```
$ test <expr>   #Comando test
$ [ <expr> ]    #Alias de test. No omitir los espacios
$ [[ <expr> ]] #Sintaxis [[ ]], solo compatible con bash
```

Para conocer las posibles expresiones de una condición, mirar la ayuda:

```
$ man test
```

Una condición se comporta como un comando cualquiera: al terminar de ejecutarse devuelve un código de error (`$?`). El código `0` se evalúa como **verdadero**, cualquier otro código se evalúa como **falso**.

ALGUNAS EXPRESIONES ÚTILES

- **Comparación léxicográfica entre cadenas de texto**: **=** (igual que), **!=** (distinto de), **'<'** (menor que), **'>'** (mayor que), **=~** (coincide con expresión regular, solo compatible con `[[`).
- **Comparación entre números**: **-eq** (igual que), **-ne** (distinto de), **-lt** (menor que), **-le** (menor o igual que), **-gt** (mayor que), **-ge** (mayor o igual que).
- **-z <valor>** (es una cadena vacía), **-n <valor>** (es una cadena no vacía).
- **-e <ruta>** (el fichero existe), **-f <ruta>** (existe y es un archivo regular).
- **-d <ruta>** (existe y es un directorio), **-L <ruta>** (existe y es un enlace simbólico).
- **-r/w/x <ruta>** (existe y tiene permiso de lectura/escritura/ejecuc. para este usuario).
- **<ruta> -ef <ruta>** (ambos archivos apuntan al mismo inodo).
- **<ruta> -nt/-ot <ruta>** (fichero es más nuevo/viejo (`mtime`) que fichero).

OPERADORES LÓGICOS

OPERADORES LÓGICOS EN `[`

- **`\( <expr> \)`**

  Fuerza una precedencia de operadores.

- **`! <expr>`**

  Niega la condición lógica.

- **`<expr> -a <expr>`**

  El resultado es el AND lógico de las condiciones.

- **`<expr> -o <expr>`**

  El resultado es el OR lógico de las condiciones.

OPERADORES LÓGICOS EN `[[`

- **`( <expr> )`**

  Fuerza una precedencia de operadores.

- **`! <expr>`**

  Niega la condición lógica.

- **`<expr> && <expr>`**

  El resultado es el AND lógico de las condiciones.

- **`<expr> || <expr>`**

  El resultado es el OR lógico de las condiciones.

OPERADORES LÓGICOS DE SHELL

El shell admite el uso de operadores lógicos entre comandos, evaluando cada comando según su código de salida: `0` es verdadero y cualquier otro valor es falso.

- **`! <comando>`**

  Niega el resultado del comando (si es verdadero, da `1` (falso); si es falso, da `0` (verdadero)).

- **`<comando> && <comando>`**

  AND lógico (si ambos son verdaderos, da `0` (verdadero); si alguno es falso, da `1` (falso)).

- **`<comando> || <comando>`**

  OR lógico (si alguno es verdadero, da `0` (verdadero); si ambos son falsos, da `1` (falso)).

OPERADORES LÓGICOS COMO CONDICIONALES

Los operadores AND (`&&`) y OR (`||`) actúan en **cortocircuito**, es decir, si al evaluar el primer comando se conoce el resultado final del operador, el segundo comando no se evalúa:

- `$ <comando1> && <comando2>`

  Se ejecutará `<comando1>`, y, solo si ha ido bien, se ejecutará `<comando2>`.

- `$ <comando1> || <comando2>`

  Se ejecutará `<comando1>`, y, solo si falla, se ejecutará `<comando2>`.

Sabiendo esto, ambos operadores se pueden usar para ejecutar comandos solo si otros comandos han terminado con éxito (o error).

Ejemplo típico:

```
# apt update && apt upgrade
```

Si `apt update` falla, no se ejecutará `apt upgrade`.

WHILE

Ejecuta comandos repetidamente mientras la condición sea cierta. Sintaxis:

```
while <condición>; do
    <comandos...>
done
```

FOR..IN

Ejecuta comandos tantas veces como elementos tenga una lista, recorrida por una variable (en cada iteración tendrá un valor de la lista). Sintaxis:

```
for <var> in <lista>; do
    <comandos...>
done
```

`<var>` es el nombre de la variable (sin el símbolo `$`), y `<lista>` puede ser cualquier string, donde sus elementos están separados por espacios, tabuladores o saltos de línea.

El comando `xargs` es, en ocasiones, una alternativa a `for..in`.

FOR (SOLO BASH)

Equivalente al clásico bucle `for` de C. Sintaxis:

```
for (( <inicialización> ; <condición> ; <paso> )); do
    <comandos...>
done
```

Primero se ejecuta `<inicialización>` una vez, evalúa `<condición>` antes de cada iteración (sale si es falsa) y ejecuta `<paso>` al final de cada iteración.

Esta sentencia solo es compatible con `bash`.

SEQ

En la sentencia `for..in` (o `xargs`), que recorre listas (strings separados por espacios), se pueden introducir listas literales (`1 2 3`) o listas generadas por otros comandos, como:

```
$ seq <fin>
$ seq <inicio> <fin>
$ seq <inicio> <paso> <fin>
```

Genera una lista desde `<inicio>` hasta `<fin>`, sumando `<paso>` cada vez.

Si `<inicio>` no se especifica, se asume 1.

Si `<paso>` no se especifica, se asume 1.

Con la opción `-w`, los números devueltos tendrán ceros a la izquierda para que todos tengan el mismo número de cifras.

Ejemplo (para generar 100 archivos):

```
$ for i in $(seq -w 100); do
$     touch "file_$i"
$ done
```

Equivale a:

```
$ touch file_001
...
$ touch file_100
```

ELEMENTOS DE UN SCRIPT, EJECUTAR UN SCRIPT, ARGUMENTOS, ENTORNO, DEBUG, FUNCIONES Y ÁMBITO

COMANDO EXEC, GETOPTS

SCRIPTS Y FUNCIONES SHELL

¿QUÉ ES UN SCRIPT?

Un **script** o *guión* es una secuencia de pasos que mandamos ejecutar al sistema. Es como ejecutar una serie de comandos seguidos, pero en un archivo y que se ejecuten automáticamente uno detrás de otro (sin nuestra intervención necesariamente).

Los scripts shell suelen usar la extensión `.sh`.

SHEBANG

Se puede establecer en cualquier script una primera línea especial llamada **shebang** que empieza por **`#!`** y le sigue el programa o comando que interpretará el actual script cuando lo ejecutemos mediante `./`. Vale para scripts de cualquier lenguaje.

Posibles shebangs de un script:

- **`#!/bin/sh`** para ejecutar un script shell compatible con POSIX (suele enlazar a dash).
- **`#!/bin/bash`** para ejecutar un script shell con funcionalidades propias de bash.
- **`#!/usr/bin/env python3`** para ejecutar un script Python. `/usr/bin/env` se usa para cuando no se sabe la ruta del intérprete (de Python en este caso) en el sistema.
- **`#!/usr/bin/awk -f`** para ejecutar un script AWK (lenguaje de búsqueda de patrones y procesado).

.BASHRC

Existen unos scripts que **se ejecutan justo al iniciarse un shell bash interactivo** sin inicio de sesión (ejemplos: abrir ventana o pestaña shell, abrir subshell, ...) y permiten definir configuraciones, cambiar el aspecto (colores, prompt, etc.), crear alias de comandos, definir funciones y variables, etc.

- `/etc/bashrc` o `/etc/bash.bashrc` (según la distribución): primer script que se ejecuta al **iniciar el shell**. Afecta a todos los usuarios del sistema.
- `~/.bashrc`: se ejecuta al **iniciar el shell**, justo después del anterior. Cada usuario tiene el suyo y puede personalizarlo.

Otros intérpretes shell usan scripts equivalentes a estos, pero con otros nombres, como `zsh` (`/etc/zshrc` y `~/.zshrc`).

.PROFILE

De la misma forma que `.bashrc`, existen otros scripts que se ejecutan al iniciar un shell bash **con inicio de sesión** (ejemplos: `SSH`, `su -`, `login`, ...):

- `/etc/profile`: primer script que se ejecuta al **iniciar sesión** cualquier usuario.
- `~/.profile` o `~/.bash_profile` (según la distribución): se ejecuta al **iniciar sesión** un usuario (cada uno tiene el suyo), justo después del anterior.

Otros intérpretes shell tienen scripts equivalentes a estos, pero con otros nombres, como `zsh` (`/etc/zprofile` y `~/.zprofile`).

EJECUTAR UN SCRIPT SHELL

Para ejecutar un script shell (llamado `script.sh`, por ejemplo):

- `sh script.sh`: usa el intérprete `/bin/sh` (se puede sustituir por `bash`, `zsh`, etc. u otros lenguajes de programación como python, perl, etc.). Lo ejecuta **en un nuevo subshell** pero no requiere permiso de ejecución.
- `./script.sh`: ejecuta el script **en un nuevo subshell** con el intérprete definido por el shebang. Requiere permiso de ejecución para el usuario actual.
- `. script.sh`: comando built-in POSIX para ejecutar un script **en el shell actual**. En bash, existe el sinónimo `source` (`source script.sh`). No requiere permiso de ejecución.

EXEC

Al ejecutar un comando en el shell, se crea un subproceso (y un subshell) que ejecutará ese comando.

Si, en su lugar, se quiere reemplazar la shell actual por un comando determinado:

```
$ exec <comando> <argumentos...>
```

ARGUMENTOS

Al ejecutar un script shell se definen automáticamente algunas variables especiales para acceder a los argumentos que se introdujeron.

$1 representa el primer argumento pasado al ejecutar el script, **$2** el segundo argumento, **$3** el tercero, ...

$0 representa el nombre del ejecutable usado para lanzar el script.

$# da el número de argumentos.

$@ muestra la lista completa de argumentos desde el primero (`$1 $2 $3` ...).

Si este es el archivo `script.sh`:

```
#!/bin/bash
echo "Has ejecutado $0 con $#
argumentos: $1, $2 y $3."
```

Al ejecutarlo:

```
$ ./script.sh 1 hola 6
Has ejecutado ./script.sh con
3 argumentos: 1, hola y 6.
```

ARGUMENTOS

Para casos en los que se quieran excluir los primeros argumentos de `$@`, existe el siguiente built-in:

```
shift [<n>]
```

Renombra los argumentos `$<n>+1`, `$<n>+2`, `$<n>+3`,... como `$1`, `$2`, `$3`, ..., de forma que descarta los `<n>` primeros y desplaza el resto `<n>` veces hacia atrás.

En caso de que `<n>` no se especifique, se asume `1`.

Para recorrer todos los argumentos:

```
for arg in "$@"; do
    ...
done
```

Para recorrer todos los argumentos desde el 4º (es decir, excluyendo los 3 primeros):

```
shift 3
for arg in "$@"; do
    ...
done
```

GETOPTS

Para buscar **opciones** en los argumentos:

```
$ getopts <ops> <var> [<args...>]
```

Obtiene de `$1` .. `$N` (o `<args...>`) opciones especificadas en `<ops>` (letras de opciones) y si encuentra alguna, la guarda en la variable `<var>`.

Si alguna letra de `<ops>` es seguida de `:`, es una opción con argumento, que guardará en la variable `OPTARG`. Si `<ops>` empieza por `:`, no imprimirá los errores.

Ejemplo `script.sh`:

```
while getopts 'ab' opt; do
  case $opt in
    a) echo "Opción -a" ;;
    b) echo "Opción -b" ;;
  esac
done
```

En el shell:

```
$ sh script.sh -b file.txt
Opción -b
```

OTRAS VARIABLES ESPECIALES

Un script, como cualquier comando ejecutable, puede leer de la entrada estándar, escribir en la salida estándar y/o en la de error y terminar con un código de error.

El código de error del último script ejecutado se puede ver en la variable `$?` (`0` si todo ha ido bien, cualquier otro número ante un error).

La variable `$!` contiene el PID del último proceso en segundo plano que ha lanzado el shell actual.

La variable `$$` contiene el PID del proceso del shell actual.

VARIABLES DE ENTORNO

Un script (y cualquier comando), al ser ejecutado en un shell, tiene acceso a las variables de entorno que hubiera vigentes en ese shell en el momento de la ejecución. También puede modificarlas, pero solo si se ejecuta sin crear un subshell (mediante `.`).

Sin embargo, un script (o comando) ejecutado en un nuevo subshell (mediante `./`) no tiene acceso a las variables de shell (variables donde no se usó `export`) del shell padre.

DEBUG

Una técnica útil para corregir errores de los scripts de shell es poner el siguiente como el primer comando del script:

```
$ set -x
```

Esto habilitará el **modo debug**, que imprimirá cada comando que ejecuta el script, precedido por el carácter +.

Para deshacer el modo debug:

```
$ set +x
```

FUNCIONES

Una función nos permite dar un nombre a un conjunto de comandos, que luego podremos invocar tan solo escribiendo su nombre.

Para definir una función:

```
<nombre_fun> () {
    <comandos...>
}
```

Para llamarla/invocarla (esto es, para ejecutar lo que hay dentro):

```
<nombre_fun>
```

Para llamarla/invocarla con argumentos:

```
<nombre_fun> <argumentos...>
```

Una función puede recibir argumentos igual que un script, y dentro de la función se referencian igual que lo hace un script (`$1`, `$2`, `$3`, ..., `$@`, `$#`).

ÁMBITOS Y VARIABLES LOCALES

Al llamar a una función, se crea un **ámbito** propio de la función.

Las sentencias que se ejecuten dentro de las funciones tienen acceso a las variables de shell y variables de entorno vistas anteriormente, pero también pueden crear sus propias **variables locales**, que, a diferencia de las otras, solo pertenecen al **ámbito** de la función y desaparecen al salir de ella.

Para invocar los comandos de la función sin crear un **ámbito** local (es decir, usando el ámbito actual), se invoca mediante el built-in `.`:

```
. <nombre_fun> [<argumentos...>]
```

ÁMBITOS Y VARIABLES LOCALES

Para declarar una **variable local**:

```
local <nombre_var>
```

Para declararla con un valor:

```
local <nombre_var>=<valor>
```

Para cambiar el valor a una variable ya declarada como local, la sintaxis es la misma que al modificar una variable de shell:

```
<nombre_var>=<valor>
```

Si se hace la anterior asignación con una variable no declarada como local, se estará modificando una variable de shell.

RESULTADO DE UNA FUNCIÓN

Para salir de una función:

```
return [<num_codigo>]
```

Si no se especifica el código de error, se usará el del último comando ejecutado en la función.

El código de error que devuelve la función se puede ver con la variable `$?`, igual que con cualquier comando.