Capítulo 2

Shell

Al terminar este capítulo, entenderás:

- Qué es una shell y por qué es tan importante.
- Cómo trabajar con procesos, trabajos y señales.
- Cómo redirigir la entrada y salida de los programas.
- Cómo gestionar usuarios, grupos, propietarios y permisos.
- Cómo instalar paquetes y configurar repositorios.
- Cómo gestionar servicios.
- Qué son los parámetros del núcleo y cómo configurarlos.

El intérprete de comandos (*shell*) es probablemente una de las herramientas más interesantes del sistema operativo GNU/Linux, y de cualquier POSIX en general. Una shell no es más que un programa interactivo que permite al usuario ejecutar otros programas manteniendo cierto control sobre ellos. Como con cualquier tecnología o herramienta que merezca la pena, es fácil encontrar grandes partidarios y detractores. Lo cierto es que para un novato la shell tiene un apariencia extraña, rudimentaria, obsoleta y de aspecto absolutamente espartano en un mundo en el que las pantallas multi-táctiles, la realidad virtual o el reconocimiento de voz son algo cotidiano.

Una interfaz de comandos —a menudo denominada CLI— es mucho menos intuitiva[1] que una GUI. Sin embargo, cuando la capacidad y el nivel de detalle (y por tanto la complejidad) de la herramienta aumentan, la interfaz gráfica resulta mucho más difícil de desarrollar, requiere tanto o más entrenamiento, y su uso suele ser menos productivo que una interfaz de comandos. Ejemplos muy evidentes de esto se pueden encontrar en programas de diseño 3D, herramientas CAD o de edición de imágenes o vídeo, con literalmente miles de opciones repartidas en innumerables menús. De hecho, no es extraño que algunos de estos programas con interfaces gráficas

[1] El manejo *intuitivo* se refiere a la posibilidad de realizar un uso productivo de una herramienta sin conocimiento o formación previa. El adjetivo se puede aplicar tanto a la herramienta como al usuario.

tan complejas ofrezcan también algún tipo de sistema de comandos para sus usuarios más avanzados, como es el caso de `AutoCAD`.

Otra buena razón por la que la interfaz de comandos puede resultar más productiva es porque te da la posibilidad de crear secuencias de comandos (*scripts*), permitiendo automatizar tareas repetitivas. Por ejemplo, cuando se quiere explicar una secuencia de acciones, incluso de complejidad media, usar comandos es mucho más sencillo que dar instrucciones para acceder a menús, botones y listas desplegables, algo que a menudo requiere grabar en vídeo la secuencia de acciones —los llamados *screencast*. En el caso de la shell, además existe todo un lenguaje de programación llamado *C-Shell* que dispone de variables, condicionales, bucles, funciones, etc., así que se puede ir mucho más lejos que una mera secuencia de comandos.

script

Un script en un archivo de texto que contiene secuencias de comandos. Estos comandos son interpretados por otro programa que lee y procesa el archivo. Hablamos de *shell scripts* cuando el programa que los interpreta es una shell.

En todo caso, sería un error subestimar o considerar anticuada una herramienta simplemente por el hecho de basarse en una interfaz de comandos, y desde luego lo es en el caso de la shell de GNU/Linux.

A menudo, las aplicaciones bien diseñadas definen un conjunto de acciones en una librería, que pueden ser utilizadas indistintamente desde *front-ends*[2] gráficos, línea de comandos, o incluso desarrollando otras aplicaciones a medida.

2.1. GNU Bash

Entre la gran variedad de shells que Los sistemas POSIX han conocido desde principios de los años 70, `Bash` es probablemente una de las más veteranas, conocidas y potentes. Aunque `bash` tiene características muy interesantes (como el autocompletado contextual), primero debemos abordar cuestiones básicas que todo usuario avanzado debería conocer.

Además de la propia shell, hay una colección de programas (agrupados bajo el nombre de GNU `coreutils`) que realizan acciones relativamente sencillas, pero que están diseñados de modo que se puedan combinar para realizar operaciones de complejidad nada trivial de forma bastante simple, una vez que se entiende un concepto clave: la redirección de la entrada/salida.

[2] Se denomina *front-end* a la capa de software que proporciona una interfaz —del tipo que sea— a un programa o librería que ofrece su funcionalidad por medio de una serie de funciones o clases (API)

Estos y otros programas diseñados por terceros, pero que respetan la misma filosofía, junto con C-Shell, proporcionan un ecosistema con infinitas posibilidades para todo tipo de tareas, que pueden ir desde la administración de sistemas hasta las más sofisticadas técnicas de *pen-testing*, pasando por el desarrollo de aplicaciones de todo tipo, incluso maliciosas. Y como la shell es la forma más habitual y flexible para administrar un sistema GNU/Linux, aprovecharemos este capítulo para aprender lo básico sobre gestión de procesos, programas, usuarios, archivos y servicios.

2.2. Valor de retorno

Los sistemas POSIX asumen que cuando un programa acaba devuelve un entero. Este valor es recogido por su proceso padre (que puede ser una shell) y ofrece información muy útil acerca del resultado de la ejecución.

Un programa que realiza su función satisfactoriamente debería devolver siempre un valor 0. Puede devolver un valor distinto para indicar que hubo un problema concreto, que su página de manual debería explicar.

Éste es el motivo por el que el estándar del lenguaje C dice que toda función `main()` ha de devolver un entero y que por defecto su valor de retorno debería ser cero. Según eso, el 'hola mundo' correcto en C es el siguiente:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    puts("hello world\n");
    return 0;
}
```

Por ejemplo, el siguiente listado muestra la ejecución del comando `ls /`, que lista el contenido del directorio raíz:

```
~$ ls /
bin   etc         lib         mnt   root  selinux  tmp  vmlinuz
boot  home        lost+found  opt   run   srv      usr
dev   initrd.img  media       proc  sbin  sys      var
```

En este caso, la shell `bash` crea un proceso en el que ejecuta el programa `ls` con el argumento `/`, que representa la raíz del sistema de archivos. Cuando el programa termina, la shell recoge el valor de retorno. Ese valor se almacena en una *variable de entorno* llamada `?`[3].

[3] Sí, el signo de interrogación

Puedes imprimir el valor de cualquier variable utilizando el comando `echo` y colocando el símbolo `$` delante del nombre de la variable. O sea que puedes ver el valor de retorno de un comando si inmediatamente después de terminar, ejecutas:

```
~$ echo $?
0
```

Si se produce un error, el programa retorna un código de error, independientemente de que el programa muestre un mensaje de error. Lo puedes comprobar en el siguiente ejemplo:

echo

Escribe la cadena de texto que se le indique como argumento, incluyendo las variables (identificadores precedidos del símbolo `$`).

```
~$ ls noexiste
ls: cannot access noexiste: No such file or directory
~$ echo $?
2
```

La documentación de cada programa indica el significado de cada uno de los posibles valores de retorno. Si consultas la página de manual de `ls` (con `man ls`) verás algo como:

```
Exit status:
    0      if OK,
    1      if minor problems (e.g., cannot access subdirectory),
    2      if serious trouble (e.g., cannot access command-line argument).
```

Las *señales* también se utilizan para indicar que un programa terminó como consecuencia de una situación anómala, por ejemplo si el SO (u otro programa) terminó («mató») el proceso. Por ejemplo, si el usuario pulsa `Control-C` mientras un programa se encuentra en ejecución bajo el control de una shell, esta le enviará la señal SIGINT (-2), y será ese el valor de retorno del programa.

2.3. Opciones

La mayoría de los programas CLI disponen de opciones que modifican su comportamiento aportando gran versatilidad. Las opciones normalmente presentan dos formas equivalentes: un guion y una letra (`-h`) o dos guiones y una palabra (`--help`).

El formato largo (con '`--`') es el adecuado cuando se quieren comandos autoexplicativos (como en este libro) o en un *script*. El formato corto es más conveniente cuando el usuario escribe comandos directamente en la consola, por el ahorro de tiempo, obviamente.

Para evitar sobrecargar las explicaciones sobre cada comando, en este capítulo vamos a obviar la utilidad precisa de cada opción. El lector puede consultar la página de manual del programa (`man comando`) o con la opción `--help` para conocer la función de cada opción.

2.4. Procesos

Un 'proceso' es una abstracción del SO para ejecutar un programa conforme a determinados parámetros de seguridad, prioridad y privilegios de acceso a recursos. Cualquier proceso puede crear procesos *hijos*, que heredan muchas de las propiedades de su *padre*, tales como privilegios y descriptores de archivos abiertos. Para listar procesos se utiliza `ps`, que es un abreviatura de *process status*. Por ejemplo, un usuario puede ver los procesos asociados al terminal al que está conectado con el comando `ps T`. Cada proceso tiene un identificador numérico único asignado por el SO, llamado PID. Es el número que aparece en la primera columna del siguiente ejemplo:

```
~$ ps T
  PID TTY          TIME CMD
 4970 pts/2    00:00:00 bash
 5107 pts/2    00:00:01 gedit
 5164 pts/2    00:00:00 bash
 6021 pts/2    00:00:01 firefox-bin
 6204 pts/2    00:00:00 ps
```

La segunda columna, TTY (TeleTYpewriter), es el terminal al que está asociado; la tercera, (TIME), el tiempo de procesador otorgado al proceso hasta el momento; y por último (CMD), el comando tal como se lanzó.

El programa `ps` dispone de una amplia variedad de opciones. Por ejemplo, la opción `f` muestra la «genealogía» de los procesos. Fíjate que las opciones de `ps` no van precesidas de guion, como es habitual. Veamos qué ocurre al ejecutar `ps f`:

```
~$ ps f
  PID TTY      STAT   TIME COMMAND
 4970 pts/2    Ss     0:00 bash
 5107 pts/2    T      0:01  \_ gedit
 5164 pts/2    S      0:00  \_ bash
 5300 pts/2    Sl     0:01      \_ /usr/lib/iceweasel/firefox-bin
 6283 pts/2    R+     0:00      \_ ps f
 4592 pts/0    Ss     0:00 bash
 4755 pts/0    S+     0:28  \_ code shell.tex
```

También aparece una columna adicional (STAT) que indica el estado del proceso. El primer carácter: R (*running*), T (*stopped*) y S (*sleep*); el segundo: s (*session leader*), l (*multi-hilo*), + (*en primer plano*).

Por ejemplo, si un proceso no responde a su interfaz gráfica (si la tiene) y el usuario no tiene control de otro modo, la operación más habitual es «matarlo» (con el comando `kill`). A pesar de su nombre, el comando `kill` sirve para enviar una señal a un proceso. La señal por defecto es SIGTERM, que efectivamente está pensada para terminar el proceso, pero se puede enviar cualquier otra señal que se especifique como argumento. Puedes ver la lista de todas las señales con `kill -l`. El siguiente comando envía la señal SIGKILL (9) al proceso con PID 5200:

```
~$ kill -9 5200
[1]+  Terminado (killed)      gedit
```

Algunas señales pueden ser ignoradas o bien capturadas por el programa para realizar acciones especificadas por el programador[4]. Este es el caso de SIGTERM, que se podría considerar una solicitud amistosa de terminación. Otras (como SIGKILL, que también termina el proceso) no pueden ser ignoradas, para así garantizar que el SO siempre tenga el control.

2.4.1. Trabajos

La shell crea un nuevo proceso hijo[5] para ejecutar cada comando que se le pide. El comportamiento normal de la shell es esperar a que ese proceso termine antes de permitir la introducción de un nuevo comando —fácil de comprobar con el programa `sleep`. Esto se llama ejecución en *primer plano* o *foreground*.

Sin embargo, en la ejecución de un programa puedes pedirle a la shell que no espere a su terminación, permitiendo incluso que el comando quede en ejecución indefinidamente. A eso se le llama ejecución en *segundo plano* o *background»* Para conseguir que la shell ejecute un comando en segundo plano basta con añadir el símbolo *ampersand* (`&`) al final de la línea de comando:

```
~$ gedit &
[1] 19777
~$
```

Como se aprecia en el listado anterior, la shell imprime una línea con dos números y luego queda disponible, a la espera de que se introduzca un nuevo comando. El primer número [entre corchetes] identifica el 'trabajo' en segundo plano (proceso hijo de la shell). El segundo número es el PID de dicho proceso, que lo identifica dentro del SO completo. Una shell puede

[4] Ejecuta `man signal para más detalles`

[5] La propia shell también se ejecuta en un proceso, que puede haber sido creado a su vez por otra shell.

ejecutar múltiples trabajos en segundo plano. La forma más sencilla de ver cuáles son es el comando `jobs`:

```
~$ firefox &
[2] 20847
~$ jobs
[1]-  Running                 gedit &
[2]+  Running                 firefox &
```

Si ejecutas el comando `fg` (*foreground*), el último comando ejecutado en segundo plano (marcado con el símbolo `+`) pasará a primer plano, bloqueando la shell. Opcionalmente admite como argumento el identificador del trabajo que quieres llevar a primer plano.

```
~$ fg %1
gedit
```

Si la shell se encuentra bloqueada en espera de la finalización de un comando en primer plano, puedes pararlo (no cerrarlo) pulsando `Control-Z` (la shell lo representa con `^Z`). Partiendo de la situación anterior:

```
~$ fg %1
gedit
^Z
[1]+  Stopped                 gedit
~$ jobs
[1]+  Stopped                 gedit
[2]-  Running                 firefox &
~$
```

Un trabajo **parado** puede volver a estado de ejecución en primer plano si ejecutas el comando `fg`, o a segundo plano si ejecutas `bg`.

```
~$ jobs
[1]+  Stopped                 gedit
[2]-  Running                 firefox &
~$ bg
[1]+ gedit &
~$
```

Los indicadores de trabajo también se pueden usar con el comando `kill` en lugar de especificar el PID.

```
~$ kill -SIGKILL %2
[4]+  Killed                  firefox
```

2.4.2. Otros modos de identificar procesos

A veces, buscar en la lista de procesos (`ps`) no es la forma más cómoda de localizar un proceso. Puede ser más sencillo utilizar su nombre (con `pidof` o `pgrep`):

```
~$ pidof firefox
105894
```

O bien lo puedes identificar por un archivo o dispositivo que el proceso tenga abierto, o con un puerto al que esté vinculado (con `fuser`):

```
~$ fuser 17500/tcp
17500/tcp:          10589
```

También se puede enviar una señal (por defecto con la intención de matarlo) con estos valores (nombre y puerto):

```
~$ pkill firefox
~$ fuser --kill 17500/tcp
17500/tcp:          10589
```

Por supuesto, estos comandos tienen opciones que les permiten buscar procesos por otros criterios (como el propietario o el orden de creación), que además se pueden combinar.

2.5. Entrada, salida y salida de error

En los sistemas POSIX los programas interaccionan con el SO únicamente por medio de archivos—o de abstracciones que ofrecen el mismo interfaz. Es decir, la lectura o escritura desde y hacia cualquier disco, pantalla, teclado, tarjeta de sonido o cualquier otro periférico, se realiza en términos de primitivas `read()`/`write()` de forma similar a un archivo.

Todo proceso —programa en ejecución— dispone, desde su inicio, de tres descriptores de archivo abiertos:

- entrada estándar (`stdin`) con el descriptor 0,
- salida estándar (`stdout`) con el descriptor 1, y
- salida de error estándar (`stderr`) con el descriptor 2.

Eso significa que cuando un programa trata de escribir algo (*p. ej.* con la función `puts()` en C o `System.out.println()` en Java) lo envía a su salida estándar. Del mismo modo, cuando el programa intenta leer o genera un mensaje de error, esas operaciones se realizan respectivamente sobre su entrada y salida de error estándar.

Normalmente, la entrada estándar está ligada al teclado, mientras que la salida y salida de error estándar están ligadas a la pantalla (teclado y pantalla se conocen comúnmente como 'consola' o 'terminal'[6]). Esta asociación entre la consola y los descriptores de archivo estándar la determina precisamente la shell y, mediante las órdenes adecuadas, el usuario puede alterar esa

[6] Aunque 'consola' y 'terminal' no significan exactamente lo mismo.

asociación para que la entrada y la salida de un programa queden ligadas a otra cosa, como un archivo en disco o incluso otro programa.

La ejecución anterior del comando `ls /` 'el listado de archivos del directorio raíz', apareció en pantalla porque su salida estándar corresponde por defecto a la consola.

2.6. Redirección

Es posible alterar la salida estándar de cualquier programa para enviarla, por ejemplo, a un archivo. Simplemente debes escribir el símbolo `>` (mayor-que) seguido del nombre del archivo:

```
~$ ls -l / > /tmp/root-files
~$
```

Repito: la 'redirección de salida' consigue que **cualquier** programa pueda enviar su salida en un archivo (o donde la shell determine) sin que el creador de ese programa tenga que haber hecho absolutamente nada para soportarlo.

`/tmp`

El directorio `/tmp` se utiliza por las aplicaciones para archivos *temporales* en los que almacenar logs o datos de sesión. El contenido de este directorio se elimina al apagar el computador.

Dos cosas han cambiado respecto a la ejecución anterior del comando `ls`: la primera es que se ha especificado la opción `-l` que hace que se muestren permisos, propietario y fecha de modificación de cada archivo o directorio. La segunda es que esta vez *nada* ha aparecido en la consola. La lista de los nombres de archivo ha sido almacenada en el archivo `/tmp/root-files`.

Puedes ver el contenido de ese archivo con el programa `cat`:

```
~$ cat /tmp/root-files
total 98
drwxr-xr-x   2 root root  4096 Aug 15 00:34 bin
drwxr-xr-x   4 root root  2048 Aug 17 22:22 boot
drwxr-xr-x  17 root root  3560 Aug 26 10:20 dev
drwxr-xr-x 193 root root 12288 Aug 25 18:29 etc
[...]
```

La redirección simple (`>`) crea siempre un archivo nuevo. Si existe un archivo con el mismo nombre, su contenido se sobrescribe. Pero existe otro tipo de redirección de salida que añade el contenido *al final* del archivo especificado. Se indica con doble mayor-que (`>>`):

`cat`

Lee el contenido de los archivos que se le indiquen como argumentos y lo escribe en su salida estándar. Si no se le dan argumentos, leerá de su entrada estándar.

```
~$ date > /tmp/now
~$ date >> /tmp/now
~$ cat /tmp/now
Mon Jul 15 12:05:47 CEST 2024
Mon Jul 15 12:05:49 CEST 2024
```

date

Escribe en su salida estándar la fecha y hora actual con la zona horaria configurada.

Volviendo al archivo `root-files`, veamos cómo se podrían filtrar sus líneas de modo que aparezcan únicamente las que contengan la cadena «Jun» (los archivos modificados en junio). Para eso puedes utilizar el comando `grep` del siguiente modo:

```
~$ grep Jun /tmp/root-files
drwxr-xr-x   6 root root  4096 Jun  6 17:58 home
lrwxrwxrwx   1 root root    32 Jun 27 13:09 initrd.img
lrwxrwxrwx   1 root root    28 Jun 27 13:09 vmlinuz
```

grep

Escribe en su salida estándar aquellas líneas que coincidan con un criterio especificado. Lee de los archivos indicados como argumentos (o de su entrada estándar en su defecto).

`grep`, como muchos otros programas, usa su entrada estándar como fuente de datos si no se le dan argumentos. Por tanto, utilizando la 'redirección de entrada' (con el carácter `<`) se puede lograr lo mismo ejecutando:

```
~$ grep Jun < /tmp/root-files
drwxr-xr-x   6 root root  4096 Jun  6 17:58 home
lrwxrwxrwx   1 root root    32 Jun 27 13:09 initrd.img
lrwxrwxrwx   1 root root    28 Jun 27 13:09 vmlinuz
```

La diferencia es que, en este caso, el comando `grep` no tiene constancia de estar leyendo realmente de un archivo en disco. Resulta irrelevante.

Si se desea almacenar el resultado obtenido en otro archivo, basta con utilizar de nuevo la redirección de salida. Es decir, es posible combinar redirecciones de entrada y salida en el mismo comando:

```
~$ grep Jun < /tmp/root-files > /tmp/Jun-files
```

El contenido de ese archivo está ordenado por los nombres de los archivos (la última columna). Veamos cómo reordenar esa lista en función del tamaño (quinta columna) utilizando el comando `sort`:

```
~$ sort --numeric-sort --key=5 /tmp/Jun-files
lrwxrwxrwx   1 root root    28 Jun 27 13:09 vmlinuz
lrwxrwxrwx   1 root root    32 Jun 27 13:09 initrd.img
drwxr-xr-x   6 root root  4096 Jun  6 17:58 home
```

Para lograr este resultado se han creado dos archivos temporales (`root-files` y `Jun-files`), pero si lo único que te interesa es el resultado final, hay una forma más sencilla y rápida de conseguir lo mismo sin necesidad de crear esos archivos: las tuberías.

sort

Ordena las líneas de los archivos que se le indiquen como argumento (o de su entrada estándar) y las escribe en su salida estándar. Se pueden especificar diferentes criterios de ordenación mediante opciones.

La tubería (*pipe*) conecta la salida estándar de un proceso directamente con la entrada de otro, de modo que todo lo que el primer programa escriba podrá ser leído inmediatamente por el segundo. Para indicar a la shell que cree una tubería se utiliza el carácter `|` (AltGr-1 en el teclado español). El comando para obtener directamente los archivos del directorio raíz modificados en junio ordenados por tamaño sería:

```
~$ ls -l / | grep Jun | sort -n -k5
lrwxrwxrwx   1 root root    28 Jun 27 13:09 vmlinuz
lrwxrwxrwx   1 root root    32 Jun 27 13:09 initrd.img
drwxr-xr-x   6 root root  4096 Jun  6 17:58 home
```

Nótese que `grep` y `sort` no tienen nombres de archivo en sus argumentos y que, por tanto, están leyendo líneas desde sus respectivas entradas estándar.

Por supuesto, es posible combinar las tuberías y la redirección en un mismo comando. En este caso, el resultado del comando anterior se almacena en un archivo, del mismo modo que se hacía con los comandos simples:

```
~$ ls -l / | grep Jun | sort -n -k5 > /tmp/root-jun-files-sorted-by-size
```

También se puede hacer redirección de salida con la captura hacia una variable. Se utiliza la sintaxis `$(comando)`. Por ejemplo, si quieres almacenar el número de archivos del directorio raíz en una variable, puedes ejecutar:

```
~$ num_files=$(ls -1 / | wc -l)
~$ echo $num_files
24
```

wc

Cuenta el número de líneas, palabras y caracteres de los archivos indicados como argumentos (o de su entrada estándar) y lo escribe en su salida estándar.

Otra forma muy potente aunque no tan habitual es la *sustitución de procesos* cuya sintaxis es `>(comando)` o `<(comando)`. Permite utilizar la salida o la entrada de un comando como si fuera un archivo. En realidad es así, aunque ocurre fuera de tu vista. La shell crea un archivo temporal en `/dev/fd/` que contiene la salida del programa. Por ejemplo, podemos comparar la lista de procesos ahora y dentro de 2 segundos con:

```
~$ diff <(ps aux) <(sleep 2; ps aux)
```

Es similar a la tubería, pero es más versátil porque permite utilizar la salida o la entrada de varios comandos en el mismo comando tal como ves en el ejemplo anterior.

diff

Compara el contenido de dos archivos línea a línea y escribe las diferencias en su salida estándar.

La salida de error también se puede redirigir. Por ejemplo, `cat` al igual que la mayoría de programas, imprime un mensaje por su salida de error si le das un archivo que no existe. Puedes redirigir a un archivo con `2>\file`

```
~$ cat no-existe
cat: no-existe: No existe el \file o el directorio
~$ cat no-existe 2> /tmp/cat-error
~$ cat /tmp/cat-error
cat: no-existe: No existe el \file o el directorio
```

Y por último, puedes redirigir un descriptor a otro, por ejemplo, la salida de error a la salida estándar. Mira este ejemplo:

```
1  ~$ echo hola > si-existe
2  ~$ cat si-existe no-existe
3  hola
4  cat: no-existe: No existe el \file o el directorio
5  ~$ cat si-existe no-existe > /dev/null
6  cat: no-existe: No existe el \file o el directorio
7  ~$ cat si-existe no-existe 2> /dev/null
8  hola
9  ~$ cat si-existe no-existe > salida 2>&1
10 ~$ cat salida
11 hola
12 cat: no-existe: No existe el \file o el directorio
```

El comando `echo` crea el archivo `si-existe` con el texto «hola». Después le pedimos a `cat` que lo imprima junto a otro que archivo que no existe. Por eso aparece el «hola» en la **línea 3** (que envía a `stdout`) y el mensaje de error en la **línea 4** (que envía a `stderr`). Vemos las dos porque por defecto ambas salidas están conectadas a la consola. En la **línea 5** redirigimos `stdout` a `/dev/null` (que la descarta), y por eso solo vemos el mensaje de error. En la **línea 7** es `stderr` la que descartamos, así que vemos solo el «hola». Por último en la **línea 9** redirigimos `stderr` a `stdout` y redirigimos esta última a un archivo `salida`, y vemos que al imprimirlo contiene las dos líneas. Aunque es contraintuitivo, la redirección `2>&1` debe ir al final al comando porque la shell evalúa las redirecciones de derecha a izquierda.

Es fácil apreciar el gran potencial de este sencillo mecanismo. Cualquier sistema POSIX (en especial GNU) dispone de una gran cantidad de pequeños programas especializados —como los que se han introducido aquí— que pueden combinarse mediante redirección para cubrir una gran variedad de necesidades puntuales de una forma rápida y eficiente[7].

2.7. Usuarios y grupos

Todo lo que ocurre en un sistema GNU/Linux depende de un usuario, que puede ser un humano o un servicio. El usuario se identifica con un número: el UID (User IDentifier). También hay grupos de usuarios, que también se identifican con un número: el GID (Group IDentifier). Cada usuario puede pertenecer a múltiples grupos y, para cada usuario, existe siempre un grupo que suele tener el mismo número.

Para simplificar el manejo de usuarios y grupos, se establece una correspondencia entre identificadores numéricos y nombres, que está almacenada en el archivo `/etc/passwd`. Cada fila de este archivo contiene el nombre del usuario, su UID, su GID, su nombre completo, su directorio personal (su *home*) y la shell que se ejecutará cuando el usuario inicie sesión. Aquí puedes ver algunas líneas de un archivo `/etc/passwd`:

```
root:x:0:0:root:/root:/bin/bash
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
alice:x:1000:1000:Alice Doe:/home/alice:/bin/bash
```

El usuario `root` es el superusuario del sistema, que tiene acceso a todos los archivos y puede ejecutar cualquier comando. Siempre tiene el UID 0. El usuario `backup` es un usuario del sistema que no tiene acceso a la consola (su shell es `nologin`). El usuario `alice` es el primer usuario normal, con su propio directorio personal (`/home/alice`) y `bash` como shell.

El archivo `/etc/group` contiene la correspondencia entre los nombres de grupo, su GID y los usuarios que pertenecen a cada uno. Aquí tienes algunas líneas de un archivo `/etc/group`:

```
root:x:0:
backup:x:34:
audio:x:29:alice,pulse
alice:x:1000:
```

Puedes ver el identificador de tu usuario y los grupos a los que pertenece con el comando `id`:

[7] La web `https://www.commandlinefu.com/` es una buena prueba de la gran versatilidad de la redirección y las capacidades de la shell.

```
~$ id
uid=1000(alice) gid=1000(alice)
↪  grupos=1000(alice),29(audio),44(video),46(plugdev),111(bluetooth)
```

Los identificadores de usuario son números a partir de 1000, mientras que los servicios y grupos del sistema tienen valores menores. Como puedes ver en el ejemplo, el grupo *audio* tiene el GID 29, y por cierto, identifica a los usuarios que pueden utilizar los dispositivos de sonido.

2.7.1. Propietarios y permisos

Todos los procesos son propiedad de algún usuario y ese usuario determina qué puede hacer el proceso, y a qué archivos y dispositivos puede acceder. Puedes ver el propietario de un proceso con la opción `u` del comando `ps`. En concreto el siguiente comando muestra todos los procesos del sistema y el propietario en la primera columna. Aquí solo incluimos algunas líneas para ilustrar el resultado. Si lo ejecutas en tu sistema, probablemente verás más de 200.

```
~$ ps aux
USER      PID %CPU %MEM  TTY     STAT START   TIME COMMAND
root        1  0.0  0.0  ?       Ss   16:55   0:01 /sbin/init splash
root        2  0.0  0.0  ?       S    16:55   0:00 [kthreadd]
alice    5021  0.0  0.0  ?       Ss   16:55   0:01 /lib/systemd/systemd --user
alice    8031  1.4  2.6  tty2  SLl+   16:56   3:44 /opt/google/chrome/chrome
alice   18950  0.0  0.4  ?      Ssl   17:31   0:07 /usr/libexec/gnome-terminal-server
alice   33354  0.4  1.3  ?     SLsl   18:33   0:38 /usr/share/code/code .
alice  196069  0.0  0.0  pts/5   Ss   21:05   0:00 bash
alice  202403  0.0  0.0  pts/5   R+   21:11   0:00 ps aux
```

Los archivos (y eso incluye los dispositivos representados como tales) también son propiedad de algún usuario y de un grupo. Ya has visto cómo listar los propietarios de archivos con el comando `ls -l`.

```
~$ ls -l /
total 98
lrwxrwxrwx   1 root root      7 sep 27  2023 bin -> usr/bin
drwxr-xr-x   3 root root   4096 may  2 18:43 boot
drwxr-xr-x  19 root root   4100 may  8 21:14 dev
drwxr-xr-x 207 root root  16384 may  8 21:14 etc
drwxr-xr-x   6 root root   4096 dic  1  2023 home
drwx------  22 root root   4096 may  4 15:00 root
-rw-r--r--   1 root root   1028 ene 19  2023 PKGBUILD
-rwx------   1 root root     27 feb 16  2024 vmlinuz
[...]
```

La tercera y cuarta columnas indican el usuario y el grupo propietario. En el ejemplo todos son del usuario `root` porque obviamente son archivos del sistema. La primera columna tiene información interesante codificada con una serie de letras y guiones:

La primera letra indica el tipo de archivo: `d` para directorios, `l` para enlaces simbólicos y - para archivos normales. Después aparecen 9 letras que tienes que interpretar como tres grupos. Expresan los permisos de lectura (`r`), escritura (`w`) y ejecución (`x`) del usuario, el grupo y del resto de usuarios (tres letras para cada uno). Si no tiene el permiso, aparece un guion (`-`).

Por ejemplo, la línea `drwxr-xr-x` de `boot` significa que es un directorio (`d`), que el propietario puede leer, escribir y ejecutar (`rwx`) el archivo y que tanto los miembros del grupo como el resto de usuarios solo pueden leer y ejecutar (`r-x`). Como es un directorio, 'leer' significa que se puede listar su contenido, 'escribir' que se puede crear o borrar archivos dentro de él y 'ejecutar' que puede acceder a su contenido.

Cuando se aplica a un archivo, 'ejecutar' significa que lo puede ejecutar como un programa. Por ejemplo, el archivo `vmlinuz` es un archivo normal (`-`) que solo puede ser leído, escrito (modificado o borrado) y ejecutado con el usuario `root`.

Cuando un usuario ejecuta un programa, el SO crea un proceso que hereda el propietario que lo ha ejecutado y el SO se encarga de que las operaciones que pueda realizar ese proceso respeten los permisos de los archivos a los que trate de acceder.

Por ejemplo, el archivo `/etc/shadow` almacena información sensible relacionada con las contraseñas de cada usuarios. Por ese motivo, solo el usuario `root` y el grupo `shadow` tienen permiso de lectura.

```
~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 2134 mar  3 12:24 /etc/shadow
```

Si intentas leerlo con un usuario normal, el SO te lo impedirá:

```
~$ cat /etc/shadow
cat: /etc/shadow: Permiso denegado
```

Aunque seas el administrador de tu propio PC, una buena práctica de seguridad es evitar utilizar el usuario `root` a menos que sea estrictamente necesario. Si ejecutas un programa malicioso como `root` le estarás dando acceso a todo el sistema, con lo que el daño que puede provocar es mucho mayor.

2.8. Paquetes

Muchas distribuciones de GNU/Linux utilizan *paquetes* para distribuir sus programas. Un paquete no es más que una colección de archivos estructurados en directorios para ser colocados en sus rutas adecuadas dentro del sistema de archivos cuando se instale.

En Debian GNU/Linux, Ubuntu u otras distribuciones derivadas, estos paquetes son archivos comprimidos con extensión `.deb`. Si descargas o dispones de uno de estos archivos, puedes ver su contenido con `dpkg`:

```
~$ dpkg -c gedit_48.1-9_amd64.deb
drwxr-xr-x root/root         0 2025-05-03 05:18 ./
drwxr-xr-x root/root         0 2025-05-03 05:18 ./usr/
drwxr-xr-x root/root         0 2025-05-03 05:18 ./usr/bin/
-rwxr-xr-x root/root     14568 2025-05-03 05:18 ./usr/bin/gedit
drwxr-xr-x root/root         0 2025-05-03 05:18 ./usr/lib/
[...]
```

Ahí puedes ver la ruta donde se va a instalar cada fichero contenido en el paquete. Son rutas relativas a la raíz del sistema de archivos. En este caso, el programa `gedit` se instalaría en `/usr/bin/gedit`.

Por supuesto, también puedes instalar el paquete:

```
~$ sudo dpkg -i gedit_48.1-9_amd64.deb
(Leyendo la base de datos ... 803908 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar .../gedit_48.1-9_amd64.deb ...
Desempaquetando gedit (48.1-9) sobre (48.1-9) ...
Configurando gedit (48.1-9) ...
```

Sin embargo, una característica muy importante de estos paquetes es que tienen especificadas sus *dependencias*, es decir, otros paquetes que deben estar instalados previamente. Puedes comprobar esas dependencias (y otra mucha información) sobre un paquete con `apt-cache`:

```
~$ apt-cache show gedit
Package: gedit
Version: 48.1-9
Installed-Size: 1319
Maintainer: Debian GNOME Maintainers <pkg-gnome-maintainers@lists.alioth.debian.org>
Architecture: amd64
Replaces: gedit-plugin-text-size (<< 48)
Depends: gedit-common (<< 49~), gedit-common (>= 48~), gsettings-desktop-schemas,
↪  iso-codes, gir1.2-gtk-3.0 (>= 3.22), gir1.2-gtksource-300, gir1.2-tepl-6 (>= 6.12),
↪  libc6 (>= 2.38), libcairo2 (>= 1.2.4), libgdk-pixbuf-2.0-0 (>= 2.22.0),
↪  libgedit-amtk-5-0
[...]
```

Como se puede ver, para muchos de estos paquetes se indica la versión necesaria. En ese ejemplo, puedes ver que se requiere una versión del paquete `libc6` que sea mayor o igual a la 2.38.

Obviamente, encontrar, descargar e instalar manualmente (con `dpkg`) todos esos paquetes (y sus respectivas dependencias) es una tarea terriblemente pesada. Afortunadamente, existe otro modo de hacerlo.

El programa `apt`[8] puede determinar, automática y recursivamente, las dependencias de un paquete, descargarlos e instalarlos en el orden correcto.

```
~$ sudo apt install gedit
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  gedit-common gir1.2-gtksource-4 libamtk-5-0 libamtk-5-common libtepl-5-0
Paquetes sugeridos:
  gedit-plugins
Se instalarán los siguientes paquetes NUEVOS:
  gedit gedit-common gir1.2-gtksource-4 libamtk-5-0 libamtk-5-common libtepl-5-0
0 actualizados, 6 nuevos se instalarán, 0 para eliminar y 172 no actualizados.
Se necesita descargar 59,0 kB/2.146 kB de archivos.
Se utilizarán 15,5 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]
```

2.8.1. Repositorios de paquetes

La pregunta obvia es: ¿Cómo sabe `apt` de dónde descargar todos esos paquetes? Las distribuciones proporcionan servidores (normalmente web o FTP) dónde ponen los archivos `.deb` están accesibles públicamente. Por ejemplo, para Debian es `https://ftp.debian.org/debian/` y para Ubuntu `https://archive.ubuntu.com/ubuntu/`. De ambos existen cientos de «copias espejo» (*mirrors*), normalmente una por país, como `https://ftp.es.debian.org/debian/`.

En estos repositorios, los paquetes se organizan en *releases* (o versiones de la distribución)[9]. En el caso de Debian, estas *releases* tienen un número de versión y un nombre asociado. Por ejemplo Debian 13 tiene el nombre *Trixie*.

En el archivo `/etc/apt/sources.list` se especifican los repositorios de paquetes y las *releases* que se quiere usar. Aquí puedes ver un ejemplo de ese archivo:

```
deb http://deb.debian.org/debian/ trixie main contrib non-free
```

Esta línea en concreto dice que es posible instalar paquetes oficiales (*main*), contribuciones de terceros (*contrib*) y software con licencias no libres (*non-free*), desde el repositorio de Debian para la versión *Trixie*.

[8] También existe `apt-get`, que es una versión de más bajo nivel más adecuada para scripts.

[9] En `https://www.debian.org/releases/` puedes encontrar las *releases* de Debian

Este archivo puede contener muchos repositorios, y también se pueden crear otros archivos con extensión `.list` dentro de `/etc/apt/sources.list.d` que tienen el mismo tipo de contenido.

Para saber qué paquetes (y versiones) están disponibles en los repositorios configurados, `apt` debe descargar una especie de índices que se encuentran allí mismo. Para eso ejecuta:

```
~$ sudo apt update
```

Es necesario hacer esto regularmente[10] porque el contenido de los repositorios cambia y se añaden nuevos paquetes o versiones.

En Debian hay siempre tres versiones activas:

stable
: Es la última versión publicada y su contenido no debería cambiar. Corresponde con *Trixie* en el momento de escribir este texto.

testing
: Contiene los paquetes que se están preparando para una futura versión estable y por tanto, cambia continuamente. En este momento se denomina *Forky* y ese será el nombre que tendrá la siguiente versión estable.

unstable
: Que tiene paquetes recién incorporados, experimentales o que tienen algunos problemas importantes para ser incluidos en una futura versión. Esta versión siempre se llama *sid*, que en realidad es el acrónimo de *Still In Development* (aún en desarrollo).

Por cuestiones de seguridad es importante que se puedan actualizar paquetes en los que se han descubierto vulnerabilidades o problemas graves, incluso aunque correspondan a una versión estable. Por eso, es conveniente tener los siguientes repositorios en el archivo `/etc/apt/sources.list`:

```
deb http://deb.debian.org/debian/ trixie-updates main
deb http://security.debian.org/debian-security trixie/updates main
```

Todo esto significa que los repositorios configurados determinan qué paquetes (y qué versiones) se pueden instalar. Si quieres tener un sistema completamente actualizado, es decir, con las ultimas versiones de todos los paquetes disponibles en esos repositorios, debes ejecutar:

```
~$ sudo apt upgrade
```

[10] Las distribuciones actuales tienen aplicaciones que lo hacen automáticamente.

Si incorporas repositorios de versiones nuevas y quieres actualizar tu sistema con ellos, tendrás que ejecutar:

```
~$ sudo apt dist-upgrade
```

Esto hará que la versión de tu sistema operativo corresponda con la versión del repositorio más actual. Eso lo puedes ver con:

```
~$ lsb_release -a
No LSB modules are available.
Distributor ID:^^IDebian
Description:^^IDebian GNU/Linux 12 (trixie)
Release:^^I13
Codename:^^Itrixie
```

También puedes saber qué versiones de un determinado paquete están disponibles en los repositorios configurados y en cuál de ellos están:

```
~$ apt-cache policy gedit
gedit:
  Instalados: 44.2-1
  Candidato:  44.2-1
  Tabla de versión:
     48.1-4 500
        500 http://deb.debian.org/debian sid/main amd64 Packages
     48.1-3 650
        650 http://deb.debian.org/debian testing/main amd64 Packages
 *** 44.2-1 700
        700 http://deb.debian.org/debian stable/main amd64 Packages
        100 /var/lib/dpkg/status
```

Esto quiere decir que no podrás instalar versiones de los paquetes que no estén en los repositorios configurados. La solución puede ser añadir un repositorio más reciente que sí lo contenga, pero teniendo cuidado porque las nuevas versiones de sus dependencias pueden afectar a otros paquetes.

2.8.2. Cómo encontrar paquetes

Es habitual necesitar un programa o una librería, pero no saber en qué paquete se encuentra. Veamos algunas formas de localizar paquetes, pensadas sobre todo para instalaciones de escritorio.

`command-not-found`

Este paquete modifica el comportamiento de la shell, de modo que al ejecutar un programa que no está instalado, en lugar de mostrar un simple mensaje «comando no encontrado», indica en que paquete se encuentra ese programa que tratas de ejecutar. Por ejemplo:

```
~$ filezilla
No se ha encontrado la orden «filezilla», pero se puede instalar con:
sudo apt install filezilla
```

apt-file

El programa `apt-file` busca en los índices de paquetes de los repositorios de tu sistema. Puedes indicar una ruta o parte de ella y muestra todos los paquetes que coincidan. En este caso sirva para cualquier fichero, no únicamente ejecutables. Es necesario que mantengas ese índice actualizado con `apt-file update`.

```
~$ apt-file find bin/filezilla
filezilla: /usr/bin/filezilla
```

dpkg -S

Si ya tienes instalado el programa, pero necesitas saber a qué paquete corresponde puedes ejecutar lo siguiente:

```
~$ which gedit
/usr/bin/gedit
~$ dpkg -S /usr/bin/gedit
gedit: /usr/bin/gedit
```

2.9. Servicios

Los servicios del sistema[11] son procesos en segundo plano bajo el control del SO. Arrancan automáticamente el iniciar el sistema y se encargan de tareas específicas: firewall, sonido, bluetooth, etc.; servidores: web, ssh, etc. u otras tareas de gestión: registro de eventos del sistema, montaje de dispositivos de almacenamiento, etc.

Estos servicios son gestionados por un *gestor de servicios* que se encarga de arrancarlos, pararlos o reiniciarlos en función de su configuración o del estado del sistema. El sistema de gestión de servicios más frecuente en GNU/Linux en la actualidad es `systemd`. Aunque ofrece muchas funciones, veamos aquí los comandos más básicos. Obviamente solo el administrador puede ejecutarlos.

systemctl status <servicio>
: muestra el estado del servicio.

systemctl stop/start/restart <servicio>
: arranca el servicio.

[11] A menudo llamados *daemons*, un término acuñado por ingenieros del MIT en los primeros años de UNIX.

`systemctl enable/disable <servicio>`
: habilita o deshabilita el servicio para que arranque automáticamente o no al iniciar el sistema.

`systemctl list-units`
: muestra el estado de todos los servicios disponibles.

Por ejemplo, puedes ver el estado del servicio `ssh` (que permite conexiones remotas al sistema) con:

```
~$ sudo systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: enabled)
     Active: active (running) since Fri 2025-08-01 15:21:30 CEST; 1 day 7h ago
 Invocation: a081694ecbb7403190db862ff993b604
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 965 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 1045 (sshd)
      Tasks: 1 (limit: 19048)
     Memory: 2.4M (peak: 3M)
        CPU: 25ms
     CGroup: /system.slice/ssh.service
             └─1045 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

ago 01 15:21:29 maine systemd[1]: Starting ssh.service - OpenBSD Secure Shell server...
ago 01 15:21:30 maine sshd[1045]: Server listening on 0.0.0.0 port 22.
ago 01 15:21:30 maine sshd[1045]: Server listening on :: port 22.
ago 01 15:21:30 maine systemd[1]: Started ssh.service - OpenBSD Secure Shell server.
```

Ahí puedes ver que el servicio está habilitado (*enabled*) y activo desde hace 1 día y 7 horas.

En distribuciones GNU/Linux más antiguas se utiliza el sistema `SysVinit` que permitía hacer una gestión similar con el comando `service`. Aunque `SysVinit` está en desuso, `systemd` ofrece comandos compatibles en las distribuciones modernas:

`service <servicio> status`
: muestra el estado del servicio.

`service <servicio> stop/start/restart`
: para, arranca o reinicia el servicio.

`service ---status-all`
: muestra el estado de todos los servicios instalados.

2.10. Parámetros del núcleo

Existen muchos parámetros de configuración del núcleo del SO que se pueden consultar y modificar en tiempo de ejecución. Esa modificación se aplica inmediatamente sin necesidad del reiniciar el sistema.

Estos parámetros o variables del sistema están disponibles directamente para el usuario en el directorio `/proc` aunque lo que contiene no son archivos al uso. Se trata de un sistema de archivos virtual (llamado *procfs*) que expone una gran cantidad de información del núcleo y de los procesos en ejecución.

En el directorio `/proc/sys` en particular, es donde puedes encontrar los parámetros de configuración, organizados en subdirectorios en función de su categoría. Por ejemplo, el directorio `/proc/sys/fs` contiene los parámetros del sistema de archivos y `/proc/sys/net` contiene configuración de la red.

Para ver el valor de uno de estos parámetros, puedes usar simplemente el comando `cat`. Por ejemplo, el parámetro `/proc/sys/fs/file-max` indica el número máximo de archivos que pueden estar abiertos al mismo tiempo.

```
~$ cat /proc/sys/fs/file-max
9223372036854775807
```

Como todos los parámetros están en `/proc/sys`, cuando se haga referencia a alguno de ellos a lo largo del libro, se indicará su ruta relativa, es decir, para referirnos a `/proc/sys/fs/file-max` hablaremos del 'parámetro del núcleo `fs/file-max`'.

Para modificar el valor de un parámetro se requieren permisos de superusuario, y consiste en escribir el valor deseado en el archivo virtual, algo que puedes hacer con el comando `echo`. Por ejemplo, el parámetro `vm/laptop\_mode` indica si el sistema está en «modo portátil». En este modo, el SO trata de ahorrar energía y optimizar el uso del disco duro. Para activarlo, basta con escribir un valor distinto de 0 en el archivo:

```
~# echo 1 > /proc/sys/vm/laptop_mode
~# cat /proc/sys/vm/laptop_mode
1
```

Existe un programa específico para manipular estos parámetros llamado `sysctl`. La siguiente consola muestra su uso para leer y modificar el parámetro `vm/laptop\_mode` de forma equivalente a lo visto previamente:

```
~# sysctl vm.laptop_mode
vm.laptop_mode = 0
~# sysctl -w vm.laptop_mode=1
vm.laptop_mode = 1
```

Sin embargo, estos cambios no son permanentes. Si se reinicia el sistema, el valor del parámetro (y por tanto el comportamiento correspondiente) volverá al valor por defecto. Para que estos cambios sean permanentes, deben añadirse en el archivo `/etc/sysctl.conf`. Para el parámetro anterior, la línea que se debe añadir es:

```
vm.laptop_mode = 1
```

Y ¿qué más?

La shell representa como ningún otro programa la esencia de la *filosofía Unix*, que se puede resumir en una cita atribuida a Doug McIlroy, uno de los ingenieros de Bell Labs que diseñó el sistema UNIX:

> *Escribe programas que…*
> *…hagan solo una cosa y la hagan bien.*
> *…que trabajen juntos.*
> *…que manejen flujos de texto, porque es una interfaz universal.*

Aparte de los programas que hemos visto en este capítulo, existen muchos otros diseñados con estos mismos principios, que pueden ser combinados con redirección para realizar tareas complejas. El anexo A incluye una lista de comandos habituales, aunque a lo largo del libro también se introducirán algunos más.