
BASH Unidad 3

Redirecciones, tuberías, filtros y control de trabajos

Tabla de contenidos

Tarjetas didácticas

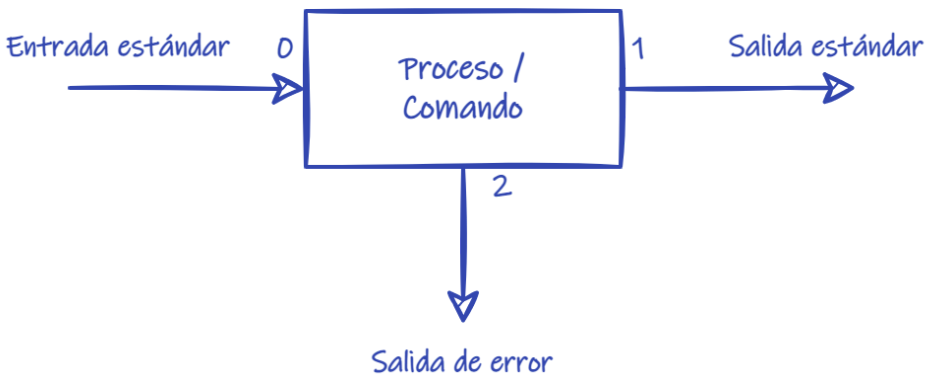
Para ayudarte a recordar los comandos de esta unidad, su uso y las opciones más comunes, puedes apoyarte en estas tarjetas didácticas. La aplicación está disponible tanto en web como para móvil o tablet.

Redirecciones de Entrada/Salida

El concepto de terminal del que hablamos en la primera parte es un poco más complejo de lo que pudiera ser a simple vista. Una terminal está formada por un dispositivo de entrada —como un teclado— y un dispositivo de salida —como una pantalla— pero ¿cómo acceden la shell y otros programas en la terminal a estos dispositivos para leer del teclado y leer de la pantalla?

Para darles acceso, el sistema operativo utiliza una abstracción. Todo proceso tiene acceso a 3 recursos: la *salida estándar*, la *entrada estándar* y la *salida de error*. Estos recursos se comportan como archivos. Los programas solo tiene que leer y escribir de estos recursos para tener acceso al teclado y la pantalla de la terminal:

- **Salida estándar**. Los programas muestran sus resultados por la terminal escribiendo en la *salida estándar*. Cuando llamamos a printf(), std::print() o usamos std::cout, estamos haciendo que se escriba en la salida estándar.
- **Entrada estándar**. Los programas leen la entrada de teclado del usuario en la terminal leyendo de la *entrada estándar*. Cuando llamamos a scanf() o usamos std::cin estamos usando la entrada estándar.
- **Salida de error**. Los programas deben mostrar sus errores en la terminal usando la *salida de error*. La salida de error también se muestra en la pantalla en la terminal, pero el mandar los mensajes de error por una vía diferente permite tratarlos de manera distinta al resto de mensajes, si fuera necesario. Cuando llamamos a fprintf(stderr, …) o usamos std::cerr estamos usando la salida de error.



Lo interesante de esta abstracción es que se puede sobreescribir. La entrada estándar para un comando puede llegar de un archivo o desde la salida de otro comando. De igual forma, tanto la salida de estándar como la de error se puede mandar a un archivo o a otro comando.

Encadenar las salidas y las entradas de los comandos es una herramienta muy potente.

Redirección de la salida estándar

Por defecto, la salida estándar de cualquier comando **envía los contenidos a la terminal, pero se puede redirigir a un archivo usan el carácter ">"**:

```
yo@mihost:~$ ls -l > lista_archivos.txt
```

En el ejemplo anterior:

- El comando ls es ejecutado y su resultado es escrito en el archivo `lista_archivos.txt`, en lugar de mostrarse por la pantalla de la terminal.
- El comando ls ve el argumento "-l" pero no ve `> lista_archivos.txt`. La redirección no es un argumento del comando. Es la shell la encargada de hacer la redirección antes de ejecutar el comando y nada en los argumentos que este recibe le indican qué es lo que está pasando, con lo que intenta imprimir por pantalla.

Cada vez que el ejemplo anterior es ejecutado, **el contenido de `lista_archivos.txt` es sobrescrito con el resultado de ls. Si lo que queremos es que los nuevos resultados se añadan al final del archivo, debemos usar en su lugar ">>"**:

```
yo@mihost:~$ ls >> lista_archivos.txt
```

En ambos casos, si el archivo no existe, es creado.

Redirección de la salida de error

La salida de error se redirecciona de forma similar a la de salida estándar, **solo que se debe añadir un 2 antes del caracter ">":**

```
yo@mihost:~$ ls -l 2> errores.txt
```

o

```
yo@mihost:~$ ls -l 2>> errores.txt
```

si queremos que la salida se añada al final del contenido del archivo `lista_archivos.txt`.

El uso de una u otra redirección depende de lo que queramos guardar en el archivo. Si queremos capturar y guardar los errores de ls, necesitamos usar la redirección de la salida

---

de error. Si queremos capturar la salida normal de ls, necesitamos usar la redirección estándar.

Obviamente, podemos usar ambas redirecciones a la vez:

```
yo@mihost:~$ ls -l > lista_archivos.txt 2> errores.txt
```

En el caso de que queramos enviar ambas salidas al mismo archivo, podemos usar una versión abreviada del comando anterior:

```
yo@mihost:~$ ls -l > lista_archivos.txt 2>&1
```

Redirección de la entrada estándar

Por defecto, la entrada estándar toma el contenido desde el teclado, pero al igual que la salida estándar, también puede ser redirigida. **Para redirigir la entrada estándar desde una archivo, en lugar de desde el teclado, se debe usar el carácter "<" de la siguiente manera**:

```
yo@mihost:~$ sort < lista_archivos.txt
```

En el ejemplo anterior usamos el comando sort para procesar el contenido de `lista_archivos.txt` y ordenarlo. El resultado se mostrará en la terminal, dado que la salida estándar del comando sort no ha sido redirigida.

Podemos redirigir también su salida de la siguiente manera:

```
yo@mihost:~$ sort < lista_archivos.txt > lista_ordenada.txt
```

Como se puede apreciar, **un comando puede tener redirigida al mismo tiempo tanto la entrada como la salida**:

- **El orden en el que se haga no importa**.
- **Los operadores "<" y ">" deben aparecer siempre después de las opciones y argumentos del comando en cuestión**, porque las redirecciones no son argumentos del comando.

Tuberías

Las redirecciones permiten conectar múltiples comandos a través de las *tuberías*. Con ellas, **los contenidos de la salida estándar de un comando pueden alimentar la entrada estándar de otro**:

```
yo@mihost:~$ ls -l | less
```

En este ejemplo, los resultados del comando ls alimentan al comando less. Esto facilita examinar un listado de archivos cuando es muy grande, usando las funciones de less.

Conectando comandos se puede hacer todo tipo de operaciones complejas. Aquí tenemos algunos ejemplos:

Ejemplos de comandos con tuberías

Comando	Resultado
ls -lt | head	Muestra los 10 archivos más nuevos en el directorio de trabajo actual. En este caso, ls lista los archivos ordenados por tiempo, mientras que head recorta para solo mostrar los 10 primeros.
du | sort -nr	Muestra una lista de directorios y cuánto espacio ocupan ordenados del más grande al más pequeño. El comando du obtiene la lista de directorios y el espacio que ocupan, mientras que sort se encarga de ordenarla.
find . -type f -print | wc -l	Muestra el número total de archivos en el directorio de trabajo actual y todos sus subdirectorios. El comando find genera la lista de archivos. Como cada archivo ocupa una línea, se usa wc para contar el número de líneas de la salida de find.

Filtros

Los filtros son comandos especializados en tomar contenidos de la entrada estándar, realizan alguna operación sobre ellos y enviar el resultado a la salida estándar. De esta manera, pueden ser combinados mediante tuberías para procesar todo tipo de información de formas muy útiles.

Ejemplos de filtros

Programa	Lo que hace
sort	Ordena la entrada estándar.
uniq	Dada una entrada ordenada, elimina las líneas duplicadas. Es decir, se asegura de que cada línea sea única.
cut	Imprime columnas que queramos de un archivo inyectado por la salida estándar.
grep	Examina cada línea que recibe de la entrada estándar e imprime por la salida estándar aquellas que contienen el patrón de caracteres especificado. La forma en la que se especifica el patrón es mediante expresiones regulares.
fmt	Lee texto desde la entrada estándar y después lo imprime formateado por la salida estándar.

---

head	Muestra las primeras líneas de la entrada estándar. Por defecto son 10 líneas, pero con la opción "-n" se puede indicar otra cifra.
tail	Muestra las últimas líneas de la entrada estándar. Por defecto son 10 líneas, pero con la opción "-n" se puede indicar otra cifra.
tr	Sustituye caracteres. Puede ser usado para hacer conversiones entre mayúsculas y minúsculas, para suprimir espacios o caracteres que se repiten.
wc	Contar las líneas, palabras y caracteres de la entrada estándar.
sed	Editor de flujo. Puede hacer transformaciones mucho más complejas que las que hace **tr**.
awk	Intérprete de un completo lenguaje especialmente diseñado para realizar acciones sobre flujos de texto.

Trabajando con filtros

Veamos un ejemplo sencillo de cómo se usan los filtros.

El fichero `/etc/passwd` contiene la información de las cuentas de usuario en el sistema. Cada usuario que quiere autenticarse en el equipo debe tener una cuenta. Y cada proceso que se ejecuta pertenece a una cuenta de usuario.

```
yo@mihost:~$ cd /etc
yo@mihost:/etc$ cat passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
...
```

Como se puede observar, cada cuenta de usuario tiene una línea y en cada línea hay distintos campos, separados por ":". ¿Cómo podríamos extraer de este archivo el nombre de todos los usuarios del sistema, ordenados de forma descendente?

Paso 1

Ante un problema, lo primero es buscar un comando o archivo que sirva de fuente de datos y examinarlo para ver cómo provee la información —que es lo que hemos hecho al ver el contenido de `/etc/passwd`—. Después tenemos que pensar cómo combinar los filtros que conocemos para obtener el resultado deseado.

Si miramos la tabla de filtros anterior, veremos que cut permite extraer columnas de un archivo. Eso es lo que queremos, extraer la primera columna de un archivo donde las columnas se separan con ":". Como es la primera vez que trabajamos con él, no sabemos qué opciones se usan para indicar la columna que se quiere extraer o si se puede indicar el carácter que se usa para separar las columnas, pero eso no es problema. Solo tenemos que consultar la página del manual de cut:

```
yo@mihost:/etc$ man cut
```

Tras hacerlo veremos que las opciones que buscamos son "-d" y "-f", donde la primera se usa para indicar el caracter separador de las columnas y el segundo permite indicar qué columna queremos extraer:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1
root
daemon
bin
sys
sync
games
man
lp
...
```

Breve inciso

Antes de continuar, supongamos que cut no tiene la capacidad de dividir las columnas usando el carácter que queramos. Es decir, no tiene opción "-d", pues siempre corta por espacios. En ese caso podríamos seguir usando cut, si antes encontramos un filtro que cambia los ":" por un espacio.

Consultado la tabla de filtro veremos que tr sirve para sustituir caracteres. Y mirando su página del manual veremos cómo se usa.

Antes:

```
yo@mihost:/etc$ cat passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
...
```

---

pero añadiendo el filtro tr:

```
yo@mihost:/etc$ cat passwd | tr ':' ' '
root x 0 0 root /root /bin/bash
daemon x 1 1 daemon /usr/sbin /usr/sbin/nologin
bin x 2 2 bin /bin /usr/sbin/nologin
sys x 3 3 sys /dev /usr/sbin/nologin
sync x 4 65534 sync /bin /bin/sync
games x 5 60 games /usr/games /usr/sbin/nologin
man x 6 12 man /var/cache/man /usr/sbin/nologin
lp x 7 7 lp /var/spool/lpd /usr/sbin/nologin
...
```

Y ya solo nos queda añadir cut con una nueva tubería, pero sin la opción "-d" porque hemos supuesto que no existe. Sin "-d", cut corta por los espacios.

```
yo@mihost:/etc$ cat passwd | tr ':' ' ' | cut -f1
root
daemon
bin
sys
sync
games
man
lp
...
```

Paso 2

Volviendo al resultado del paso 1:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1
root
daemon
bin
sys
sync
games
man
lp
...
```

Lo que falta es ordenar los nombres de usuario en orden alfabético. Nuevamente, consultamos la tabla y descubrimos que el comando sort es el que buscamos. Y mirando en su página de manual veremos que por defecto ordena de forma ascendente:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | sort
_apt
backup
bin
daemon
games
gnats
irc
jesus
...
```

pero que con la opción "-r" ordena de forma descendente.

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | sort -r
www-data
uuidd
uucp
tss
tcpdump
systemd-timesync
systemd-resolve
systemd-network
...
```

grep

Si en lugar de ordenar quisiéramos encontrar todas las cuentas que siguen cierto patrón, podríamos usar el comando grep. Este filtro lee la entrada estándar línea a línea y la compara con un patrón. Solo las líneas que encajen —o las que no, si así se lo indicamos mediante la opción "-v"— son mostradas por la salida estándar. Por ejemplo:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | grep systemd
systemd-timesync
systemd-resolve
systemd-network
...
```

Con grep el patrón de la búsqueda se define mediante un lenguaje llamado expresiones regulares. La idea es similar a los comodines de la shell, pero más complejo y potente. Por ejemplo, el comando anterior dejaría pasar cualquier línea donde aparezca la cadena “systemd” en cualquier posición del texto de la línea. Si solo queremos líneas empiecen por “systemd”, utilizaríamos el carácter especial "^" que encaja con el comienzo de línea —mientras que "$" encaja con el final de línea—.

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | grep ^systemd
systemd-timesync
```

```
systemd-resolve
systemd-network
...
```

Patrones básicos de expresiones regulares

Patrón	Lo que hace	Ejemplo
`^`	Inicio de línea	`^root` (líneas que empiezan con "root")
`$`	Final de línea	`bash$` (líneas que terminan con "bash")
`.`	Cualquier carácter	`r.ot` (root, rlot, r8ot, etc.)
`*`	Cero o más del carácter anterior	`ro*t` (rt, rot, root, rooot, etc.)
`[]`	Cualquier carácter entre corchetes	`[abc]` (a, b, o c)
`[^]`	Cualquier carácter excepto los de los corchetes	`[^abc]` (cualquier cosa, excepto a, b, o c)
`+`	Uno o más del carácter anterior	`ro+t` (rot, root, rooot, etc.)
`?`	Cero o uno del carácter anterior	`colou?r` (color o colour)
`{n}`	Exactamente n repeticiones del carácter anterior	`o{3}` (exactamente 3 "o")
`{n,m}`	Entre n y m repeticiones del carácter anterior	`o{2,4}` (entre 2 y 4 "o")
`|`	O lógico (alternativas)	`root|admin` (root o admin)
`( )`	Agrupación	`(root|admin)user` ("rootuser" o "adminuser") `(root)*` (cero o más "root") `(admin){3}` (exactamente 3 "admin")
`[0-9]`	Cualquier dígito	`user[0-9]` (user1, user2, etc.)
`[a-z]`	Cualquier letra minúscula	`[a-z]+` (una o más letras minúsculas)
`[A-Z]`	Cualquier letra mayúscula	`[A-Z]+` (una o más letras mayúsculas)

\	Desactiva el significado especial del siguiente carácter	\$ (un dolar "$") \((un paréntesis "(")

🚨Algunos de estos caracteres como {}, (), +, ? y | también tienen significado especial en BASH, por lo que es buena idea poner el patrón entre comillas simples en la línea de comandos de grep para evitar que la shell los interprete. Por ejemplo:

```
grep 'ro+t' archivo
```

o

```
grep 'user[0-9]\{2\}' archivo.
```

Puedes obtener más información sobre las expresiones regulares en grep, en la documentación.

sed

Mientras que grep solo permite buscar, sed es una herramienta mucho más compleja que permite las acciones típicas de cualquier editor: buscar, reemplazar, insertar o borrar. La diferencia con un editor como Vi es que no funciona de forma interactiva. Simplemente, toma la entrada estándar, aplica las transformaciones indicadas en la línea de comandos y muestra el resultado por la salida estándar.

Como se trata de una herramienta compleja, no vamos a verla en detalle, pero si vamos a mostrar como se puede usar para sustituir una cadena de texto o un caracter por otro, ya que es una de las aplicaciones más comunes de esta herramienta:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | sort -r | sed
"s/^uu/123/"
www-data
123idd
123cp
tss
tcpdump
systemd-timesync
systemd-resolve
systemd-network
...
```

El argumento entre comillas del comando sed indica la operación a realizar. En este "s" de sustituir. El formato de la operación es:

```
s/<expresión regular a buscar>/<cadena por la que sustituir>/
```

Por eso el comando sed sustituye los "uu" al principio de cada línea por la cadena "123".

---

awk

El comando awk es otro comando complejo. Básicamente, es el intérprete de un lenguaje de programación diseñado para recorrer documentos de texto línea a línea, seleccionar algunas mediante un patrón, dividir las líneas en columnas y realizar distintas operaciones sobre estas.

Como reemplazo de cut

El ejemplo más sencillo es como reemplazo de cut, para extraer columnas de un texto:

```
yo@mihost:/etc$ cat passwd | tr ':' ' ' | awk '{print $1}'
root
daemon
bin
sys
sync
games
man
lp
...
```

El comando awk divide cada línea automáticamente usando los espacios para delimitar los campos. En el ejemplo se imprime la primera columna indicando #1, pero se pueden elegir otras usando $2, $3, etc. y toda la línea indicando $0.

En el ejemplo anterior hemos usado el comando tr para convertir los ':' en espacios porque ese es el delimitador que awk utiliza por defecto. Pero podemos indicarle directamente a awk que utilice ':' empleando la opción '-F':

```
yo@mihost:/etc$ awk -F: '{print $1}' passwd
root
daemon
bin
sys
sync
games
man
lp
...
```

Filtrado

Antes del 'print' entre llaves —que es la acción que se ejecutará para cada línea— se puede indicar un patrón con el que seleccionar solo las líneas que cumplan con cierto criterio. Por ejemplo, si solo queremos mostrar los usuarios con identificador mayor que 100,

---

necesitamos añadir una comparación en la columna $3 del archivo `/etc/passwd`, que es la que contiene el identificador:

```
yo@mihost:/etc$ awk -F: '$3 > 100 {print $1}' passwd
nobody
systemd-resolve
messagebus
systemd-timesync
syslog
_apt
tss
uuidd
tcpdump
...
```

También podemos saltarnos las primeras N líneas, haciendo una comparación sobre la variable NR, que siempre guarda el número de la línea de entrada que se está procesando:

```
yo@mihost:/etc$ awk -F: 'NR > 2 {print $1}' passwd
bin
sys
sync
games
man
lp
...
```

También podemos filtrar líneas usando expresiones regulares —como las que vimos en el apartado sobre expresiones regulares en grep—. AWK permite usar el operador "~" para hacer coincidir una columna (o toda la línea) con un patrón de expresión regular, y el operador "!~" para las líneas que NO coincidan con el patrón. Por ejemplo, si queremos mostrar solo los usuarios cuyo nombre empiece por "sys":

```
yo@mihost:/etc$ awk -F: '$1 ~ /^sys/ {print $1}' passwd
sys
sync
syslog
systemd-timesync
systemd-resolve
systemd-network
...
```

O si queremos mostrar usuarios que NO tengan "nologin" en su shell (última columna):

```
yo@mihost:/etc$ awk -F: '$7 !~ /nologin/ {print $1, $7}' passwd
root /bin/bash
sync /bin/sync
jesus /bin/bash
...
```

Programas AWK de múltiples líneas

El comando awk admite programas de varias líneas, donde cada línea tiene un patrón y a continuación, entre llaves, la acción que se ejecutará para las líneas que encajen con el patrón correspondiente:

```
awk [opciones] '
condición1 { acción1 }
condición2 { acción2 }
...
' archivo
```

Bloques BEGIN y END

AWK proporciona dos condiciones especiales que se ejecutan en momentos específicos durante el procesamiento:

- **BEGIN**: Se ejecuta una sola vez antes de procesar cualquier línea del archivo
- **END**: Se ejecuta una sola vez después de procesar todas las líneas del archivo

Estos bloques son muy útiles para inicialización, cálculos finales y generación de informes.

Ejemplos con BEGIN

El bloque BEGIN es útil para imprimir encabezados, inicializar variables o configurar el formato de salida:

```
yo@mihost:/etc$ awk -F: 'BEGIN {print "Lista de usuarios del
sistema:"} {print $1}' passwd
Lista de usuarios del sistema:
root
daemon
bin
sys
...
```

También se puede usar para inicializar contadores:

```
yo@mihost:/etc$ awk -F: '
BEGIN {contador=0}
$3 > 100 {contador++; print $1}' passwd
nobody
```

```
systemd-resolve
messagebus
...
```

Ejemplos con END

El bloque END es perfecto para mostrar totales, promedios o resúmenes. Por ejemplo, podemos mostrar el total en el contador del ejemplo anterior:

```
yo@mihost:/etc$ awk -F: '
BEGIN {contador=0}
$3 > 100 {contador++; print $1}
END {printf "Usuarios totales (UID > 100): %d\n", contador}'
passwd
nobody
systemd-resolve
messagebus
...
Total usuarios con UID > 100: 15
```

Mientras que este sería un ejemplo más complejo dónde se cuentan diferentes tipos de usuarios:

```
yo@mihost:/etc$ awk -F: '
BEGIN {
  print "Análisis de usuarios del sistema"
  print "================================="
  usuarios_sistema = 0
  usuarios_normales = 0
}
$3 < 1000 { usuarios_sistema++ }
$3 >= 1000 { usuarios_normales++ }
END {
  printf "Usuarios del sistema: %d\n", usuarios_sistema
  printf "Usuarios normales: %d\n", usuarios_normales
  printf "Usuarios totales: %d\n",
    usuarios_sistema + usuarios_normales
}' passwd
```

Más sobre AWK

Para profundizar en el lenguaje AWK y conocer sus capacidades más avanzadas, puedes consultar este excelente recurso: «El lenguaje de procesamiento de patrones AWK y su sucesor GAWK».

---

printf

En los ejemplos anteriores verás que en el bloque END usamos la sentencia **printf**, en lugar de **print**:

```
END {
  printf "Usuarios totales (UID > 100): %d\n", contador
}
```

La sentencia **printf** proviene de C y ha sido implementada en muchos otros lenguajes; incluyendo C++, Perl, AWK, Java, PHP y como comando de BASH. **printf** es muy similar a **print** —con la salvedad de que no incluye un salto de línea al final si no se indica implícitamente— y se usa para mostrar por la salida estándar contenidos formateados de acuerdo a una cadena de formato que se le debe proporcionar. Por ejemplo:

```
printf "Apellido: %s\nNombre: %s\n", "Pedro", "Afonso"
```

Muestra por la salida estándar:

```
Apellido: Pedro
Nombre: Afonso
```

La sentencia de ejemplo:

- **`"Apellido: %s\nNombre: %s\n"` es la cadena de formato**.
- Al encontrar el primer "`%s`" en la cadena de formato, **printf** lo sustituye por "Pedro", el primer argumento después de dicha cadena.
- Al encontrar el segundo "`%s`" en la cadena de formato, **printf** lo sustituye por "Alfonso", el segundo argumento después de dicha cadena.
- **La especificación de formato "`%s`" le indica a printf que debe tratar los argumentos como cadenas** y, por tanto, que los debe formatear teniendo eso en cuenta.

Los especificadores de formato están siempre formados por el carácter "%" seguido de una letra. La lista completa se puede extraer de la documentación de AWK —GNU Awk User's Guide - Control Letters— aunque a continuación mostramos los más relevantes:

Especificador	Descripción

`%d`, `%i`	Imprime un número entero.
	`printf "Número: %d\n", 65`
	`Número: 65`

`%f`	Imprime un número en notación de punto flotante.

```
printf "Número: %f\n", 6.5
Número: 6.500000
```

`%o` Imprime un número entero en notación octal.

```
printf "Número: %o\n", 65
Número: 101
```

`%s` Imprime una cadena.

`%u` Imprime un número entero sin signo.

`%x, %X` Imprime un número entero en notación hexadecimal

```
printf "Número: %x\n", 65
Número: 41
```

`%%` Imprime un "%".

Los especificadores puende ir acompañados de modificadores, de forma que el formato general de los especificadores es el siguiente:

```
%[flag][ancho][.precisión]especificador
```

El significado de los campos depende del **especificador**, que es una de las letras que vimos en la tabla anterior.

Flag permite hacer pequeñas modificaciones en el formato, como indicar si se debe imprimir siempre el signo o si hay que rellenar con espacios en lugar de con 0. Los detalles acerca de estos flags se pueden extraer de la documentación de awk —GNU Awk User's Guide - Format Modifiers— pero, a continuación, veremos algunos ejemplos:

Modificador	Descripción

`+`	Siempre muestra el signo (+ o -)
	`printf "Número: %+d\n", 42`
	`Número: +42`

`-`	Alineación a la izquierda.
	`printf "|%-10s|\n", "hola"`
	`|hola |`

`0`	Rellena con ceros en lugar de espacios.
	`printf "Número: %05d\n", 42`
	`Número: 00042`

`#`	Prefijo alternativo (0x para hex, 0 para octal)
	`printf "Número: #x\n", 255`
	`Número: 0xff`

Ancho indica el número mínimo de caracteres del campo. Mientras que **precisión** establece el número de decimales de los números en coma flotante. En caso de que el campo sea una cadena, la precisión indica el tamaño máximo de la misma en bytes. Por ejemplo:

```
printf "|%10s|\n", "hola"         # |      hola|
printf "|%-10s|\n", "hola"        # |hola      |
printf "|%10.3f|\n", 3.14159      # |     3.142|
printf "|%-10.3f|\n", 3.14159     # |3.142     |
```

Filtros con redirecciones

Las tuberías se pueden combinar con redirecciones. Solo tenemos que recordar que cada comando de la tubería se separa por "|", por lo que cualquier redirección que queremos que afecta a un comando debe hacerse antes del "|" al final del comando.

Por ejemplo, la salida de toda la tubería se puede mandar al archivo `usuarios.txt`:

```
yo@mihost:/etc$ cat passwd | cut -d: -f1 | sort -r > usuarios.txt
```

Además, cut puede recibir el archivo `/etc/passwd` mediante una redirección de entrada, sin necesitar cat y una tubería:

```
yo@mihost:/etc$ cut -d: -f1 < passwd | sort -r > usuarios.txt
```

Y podríamos mandas los posibles errores de cut y sort a un archivo de registro de errores:

```
yo@mihost:/etc$ cut -d: -f1 < passwd 2>> errores.log | sort -r >
usuarios.txt 2>> errores.log
```

Los filtros como comandos normales

Los filtros son comandos convencionales. Aunque se hayan diseñado para aprovecharlos de forma combinada, nada nos obliga a utilizarlos en una tubería. Todos los filtros que hemos comentado esperan leer el texto sobre el que tiene que trabajar desde la entrada estándar. Pero, al mismo tiempo, todos aceptan que indiquemos, directamente, el nombre del archivo de entrada como un argumento.

Por ejemplo, cut puede cortar así:

```
yo@mihost:/etc$ cut -d: -f1 passwd
```

donde no estamos pasando el contenido de `/etc/passwd` mediante una redirección de entrada estándar, sino que se le pasa el nombre del archivo directamente como argumento.

El comando cut lo abrirá, leerá el archivo e ignorará completamente cualquier cosa que pudiera llegar por la entrada estándar.

Obviamente, podemos seguir ordenando la salida de cut con sort:

```
yo@mihost:/etc$ cut -d: -f1 passwd | sort -r
```

Pero si no quisiéramos ordenar la salida de cut, sino directamente el archivo `/etc/passwd`, podríamos usar sort directamente:

```
yo@mihost:/etc$ sort -r passwd
```

PowerShell

En los sistemas Windows se puede instalar BASH de múltiples maneras, sin embargo, la shell por defecto es PowerShell, que presenta algunas importantes diferencias filosóficas respecto a BASH. Para conocer más, puedes leer el siguiente artículo: «PowerShell vs. Bash para usuarios de Linux».

Control de trabajos

Como muchos sistemas multitarea, Linux es capaz de ejecutar múltiples procesos al mismo tiempo. Por lo que hay un grupo de comandos diseñados expresamente para controlar esos procesos:

- ps — listar los procesos que se están ejecutando en el sistema.
- kill — mandar una señal a uno o más procesos (lo que generalmente lo "mata")
- jobs — otra forma de listar tus propios procesos.
- bg — poner un proceso en segundo plano.
- fg — poner un proceso en primer plano.

Grupos de procesos y sesiones

Los procesos se agrupan en sesiones y en grupos de procesos. Todo proceso en el sistema pertenece exactamente a una sesión y a un grupo de procesos.

Cuando un usuario se autentica en la consola del sistema o abre una terminal en el entorno gráfico, se crea una sesión y la shell que se inicia se convierte en su líder. Esto es sencillo de comprobar porque el identificador de sesión es igual al identificador del proceso que lidera la sesión.

Todos los procesos descendientes de la shell que lidera la sesión pertenecen a la misma sesión. Toda sesión puede tener una terminal de control, de la que los procesos en la sesión

---

reciben señales cuando se cierra la terminal o cuando el usuario pulsa CTRL+C. Por defecto, la terminal de control es aquella en la que se crea el líder de sesión.

Cada sesión tiene uno o varios grupos de procesos, pero cada grupo de procesos solo puede pertenecer a una única sesión. Por norma, el identificador de un grupo de procesos coincide con el identificador del primer proceso del grupo.

En aquellas shell que no soportan control de trabajos, todos los procesos que crea pertenecen a la misma sesión y grupo de procesos. Mientras que en shell con control de trabajos —como BASH— a cada comando se le asigna un grupo de procesos. Es decir, que todos los procesos en el siguiente comando se ejecutan en el mismo grupo de procesos, que es diferente al de comandos posteriores y anteriores:

```
yo@mihost:~$ ls -a | sort -u | wc
```

Una forma sencilla de verlo puede ser incluir **ps -j** en el comando:

```
yo@mihost:~$ ls -a | sort -u | ps -j
  PID  PGID   SID TTY          TIME CMD
   10    10    10 pts/1    00:00:00 bash
   42    42    10 pts/1    00:00:00 ls
   43    42    10 pts/1    00:00:00 sort
   44    42    10 pts/1    00:00:00 ps
```

Como se puede ver, en todos los procesos el identificador de sesión SID es 10, que coincide con el PID del proceso "bash"; que es el proceso líder y shell de la sesión. Mientras que "ls", "sort" y "ps" están en el grupo de procesos con identificador PGID 42; exactamente el mismo identificador que el PID de "ls", que es el primer proceso del comando. La terminal de control de la sesión es "pts/1" y todos los procesos están en ella.

Hay dos tipos de grupos de procesos: los de primer plano y los de segundo plano.

- En una misma sesión solo puede haber un grupo de procesos de primer plano. Este grupo tiene asociada la terminal de control de la sesión. Las señales que informan que el usuario ha pulsado combinaciones de teclas especiales —como CTRL+C o CTRL+Z— se envían a los procesos del grupo de primera plano. Estos procesos también pueden leer la entrada estándar de la terminal, si lo desean.
- El resto de grupos de procesos en una misma sesión son de segundo plano. Estos procesos no reciben las señales mencionadas anteriormente. Y si intentan leer la entrada estándar, reciben una señal que detiene su ejecución. Los procesos solo pueden leer la entrada estándar de la terminal cuando están en el grupo de procesos de primer plano.

El mecanismo de control de trabajo de la shell crea grupos de procesos dentro de la sesión, puede detenerlos y convertirlos en grupos de primer o segundo plano.

Ejemplo práctico

Prácticamente, todos los programas con interfaz gráfica pueden lanzarse desde la línea de comando. Veamos un ejemplo con **xload**, un pequeño programa que acompaña al sistema X Window —aun la base más común del entorno gráfico de los sistemas Linux— y que muestra una representación gráfica de la carga del sistema:

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

Obsérvese que la ventana aparece, pero la shell no nos ofrece el símbolo del sistema para que podamos introducir un nuevo comando. Esto es así porque la shell está esperando a que el programa termine. Si cerramos la ventana de **xload**, el programa terminará y la shell nos mostrará el símbolo del sistema.

Poniendo un comando en segundo plano (background)

Ahora **vamos a lanzar "xload" pero en segundo plano. Eso se hace poniendo el caracter "&" al final de la línea de comandos**:

```
yo@mihost:~$ xload &
[1] 1223
yo@mihost:~$
```

En este caso la shell sí nos muestra el símbolo del sistema, porque nunca espera a que los comandos en segundo plano terminen antes de hacerlo.

Ahora supongamos que estamos en el primer caso. **No hemos usado "&" pero queremos poner el comando en segundo plano**. En ese caso:

1. **Pulsamos la combinación de teclas CTRL+Z**, haciendo que el comando en primer plano quede suspendido. Es decir, el proceso en cuestión sigue existiendo, pero permanece dormido, por lo que la shell nos mostrará el símbolo del sistema para que podamos lanzar otro programa.
2. **Mandamos el comando al segundo plano usando el comando bg**.

```
yo@mihost:~$ xload
[2]+  Detenido                 xload
yo@mihost:~$ bg
[2]+ xload &
```

Hay que tener en cuenta que, aunque en este ejemplo cada comando contiene un solo programa, en general un comando puede estar compuesto por varios programas. Las shell los ejecuta todos como procesos dentro del mismo grupo de procesos y, por tanto, pasan a primer o segundo plano juntos.

```
yo@mihost:~$ ls -a | sort -u | wc
```

```
[1] 1523
```

Listando procesos

Ahora que hay comandos en segundo plano, puede ser muy útil obtener una lista de los procesos que hemos lanzado. Para hacerlo podemos utilizar tanto el comando jobs como el comando ps.

```
yo@mihost:~$ jobs
[1]+  Ejecutando              xload &
yo@mihost:~$ ps
PID TTY TIME CMD
1211 pts/4 00:00:00 bash
1246 pts/4 00:00:00 xload
1247 pts/4 00:00:00 ps
```

El comando "ps" por defecto solo muestra los programas ejecutados por el usuario actual en la terminal desde la que es invocado. Sin embargo, permite elegir diferentes criterios para escoger los procesos que deben ser listados, así como la información que mostrar de cada uno. Obviamente, esto se hace mediante opciones, de las cuales las de uso más común son las de los siguientes ejemplos:

Ejemplos del comando ps

Comando	Resultado
ps a	Elimina la restricción de solo mostrar los procesos del usuario actual.
ps x	Elimina la restricción de solo mostrar los procesos lanzados desde el terminal actual.
ps ax	Lista los procesos de todos los usuarios en todas las terminales.
ps -A	Lista todos los procesos del sistema.
ps -j	Formato con información de la sesión y el grupo de procesos.
ps -p 1287	Muestra solo el proceso con el identificador (PID) indicado.
ps -u yo	Muestra solo los procesos del usuario "yo".
ps u	Activa un formato de salida más orientado al usuario que el formato por defecto. Por ejemplo, **ps aux**.
ps auxf	La opción "f" permite el uso de arte ASCII para destacar los detalles de la jerarquía de procesos. Es decir, qué procesos son hijos de otros procesos.

---

ps -ouser,pid,ppid,cmd	La opción "o" permite elegir, con una lista de campos separada por comas, la información que se desea ver de cada proceso.
ps -A --no-headers	Muestra todos los procesos, pero omite el encabezado de la tabla donde se indica el contenido de cada columna. La opción "--no-headers" es interesante si se va a procesar posteriormente la lista de procesos de alguna manera.

Devolviendo un comando al primer plano

Al ejecutar el comando **"jobs" obtenemos una lista numerada de los comandos lanzados** desde la terminal actual:

```
yo@mihost:~$ xload &
[1] 1761
yo@mihost:~$ xload &
[2] 1762
yo@mihost:~$ xload &
[3] 1779
yo@mihost:~$ jobs
[1]   Ejecutando              xload &
[2]-  Ejecutando              xload &
[3]+  Ejecutando              xload &
```

Estos *números de trabajo* se pueden usar tanto con los comandos bg como fg para indicar qué comando queremos enviar al segundo o al primer plano, respectivamente. Por ejemplo, si queremos traer al primer plano el trabajo 2:

```
yo@mihost:~$ fg 2
xload
```

Nótese que como **ahora que tenemos un comando "xload" en el primer plano, la shell no nos muestra el símbolo del sistema, ya que para hacerlo está esperando a que dicho comando termine**.

Si con los comandos **fg** y **bg** no se especifica un número de trabajo, simplemente actuarán sobre el último trabajo detenido —mediante CTRL+Z, por ejemplo— o enviado al segundo plano.

Matando procesos

Supongamos que tenemos un programa que deja de responder y queremos eliminarlo. Entonces debemos usar kill para terminar el proceso correspondiente y liberar sus recursos:

---

1. **Lo primero sería identificar el proceso que se quiere matar**. Para eso se puede usar tanto **jobs** como **ps**.
2. **Después se utilizaría dicho identificador junto al comando "kill"**:
    a. Si hemos usado **jobs**, tendremos un número de trabajo que se usaría así: `kill %numero_del_trabajo`
    b. Si hemos usado **ps**, tendremos el identificador del proceso (PID), que se usaría así: `kill pid_del_proceso`

Veamos un ejemplo:

```
yo@mihost:~$ xload &
[1] 1292
yo@mihost:~$ jobs
[1]+ Ejecutando              xload &
yo@mihost:~$ kill %1
yo@mihost:~$ xload &
[2] 1293
[1]   Terminado              xload
yo@mihost:~$ ps
PID TTY TIME CMD
1280 pts/5 00:00:00 bash
1293 pts/5 00:00:00 xload
1294 pts/5 00:00:00 ps
yo@mihost:~$ kill 1293
yo@mihost:~$ xload &
[3] 1295
[2]   Terminado               xload
yo@mihost:~$
```

Aunque estamos **usando "kill" para matar procesos, su verdadero propósito es enviar *señales* a estos**. La mayor parte del tiempo las señales son enviadas con el objeto de indicar el proceso que debe terminar, pero también tienen otras funciones.

Los programas bien hechos suelen estar pendientes de las señales del sistema operativo y responden a ellas. En la mayor parte de los casos, lo que hacen estos programas cuando reciben una señal es terminar de forma adecuada y segura. Por ejemplo, un editor de texto debe escuchar cualquier señal que indique que el usuario ha terminado su sesión o que el ordenador va a apagarse. Al recibirla, debe salvar el documento actual y terminar.

El comando **kill** puede enviar todo tipo de señales. Solo con ejecutar el comando:

```
yo@mihost:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1      11) SIGSEGV     12) SIGUSR2
```

```
 13) SIGPIPE     14) SIGALRM     15) SIGTERM...
```

se obtiene una lista de todas las señales soportadas. Algunas de las más útiles son:

Señal nº	Nombre	Descripción
1	SIGHUP	Esta señal se envía a los procesos de una sesión cuando se cierra la terminal o se pierde la conexión. Primero se envía al líder de sesión. Si este muere, se manda a todos los procesos en los grupos de procesos que hayan quedado huérfanos.
2	SIGINT	Indica a los procesos que deben interrumpirse. Esta señal es enviada a los procesos del grupo de primer plano al pulsar la combinación de teclas CTRL+C mientras se está ejecutando en una terminal.
15	SIGTERM	Indica los procesos que deben terminar. Esta es la señal enviada por defecto por el comando **kill** si no se especifica otra señal.
9	SIGKILL	Esta señal causa la muerte instantánea del proceso que la recibe de manos del núcleo de Linux. Así que los procesos no pueden estar pendientes de esta señal.

Así que **si tenemos un proceso que está colgado y queremos terminarlo**:

1. **Se usa "ps" para obtener el identificador del proceso (PID) en cuestión**.
2. **Se envía una señal SIGTERM a ese PID**.
3. **Si el proceso no termina, se envía una señal un poco más dura (SIGKILL) para que lo haga**.

```
yo@mihost:~$ ps x | grep programa_colgado
PID TTY STAT TIME COMMAND
...
2931 pts/5 SN 0:00 programa_colgado
...
yo@mihost:~$ kill 2931
yo@mihost:~$ kill -SIGKILL 2931
```

En el ejemplo anterior:

1. Usamos el comando **ps** con la opción "x" para listar nuestros procesos —incluso los que no se han lanzado desde la terminal actual—. La salida de **ps** es entubada en el comando **grep** para filtrar y solo listar el programa que nos interesa.
2. Después, usamos **kill** para enviar una señal SIGTERM al programa problemático.
3. Finalmente, como el programa no termina, le enviamos la señal SIGKILL.

Esto último también podíamos haberlo hecho indicando el número de la señal (9) en lugar de su nombre (SIGKILL):

```
yo@mihost:~$ kill -9
```

Lo que en ocasiones puede resultar más cómodo.

Tarjetas didácticas

Para ayudarte a recordar los comandos de esta unidad, su uso y las opciones más comunes, puedes apoyarte en estas tarjetas didácticas. La aplicación está disponible tanto en web como para móvil o tablet.
