Unidad 5

Programación Shell Script Linux

Administración de sistemas operativos

5

Índice

Introducción

Una de las mayores causas del auge de los sistemas Unix/Linux es que sirven muy eficientemente como plataforma para el desarrollo de los sistemas.

Una de las principales características que tienen los sistemas Linux, por no decir la principal, es que cuentan con más de 200 comandos básicos propios del sistema. Estos comandos nos permiten realizar prácticamente cualquier acción en el sistema. De hecho, es gracias a esto por lo que podemos tener distribuciones de Linux sin interfaz gráfica. No obstante, lo mejor de todo esto es que la gran mayoría de los comandos se pueden combinar entre sí de manera muy sencilla a la vez que eficiente.

El Shell de Linux es un programa del sistema que se encarga de leer los comandos y convertirlos en lenguaje entendible para el sistema. Además, este propio Shell además de tener la función de interprete de comandos, también se puede usar como un lenguaje de programación al completo, lenguaje que veremos en esta unidad.

Al finalizar esta unidad

- Conoceremos los comandos básicos de Linux.
- Sabremos como emplear de forma correcta las distintas estructuras del lenguaje de programación que proporciona el Shell de Linux.
- Seremos capaces de redireccionar la entrada y la salida estándar tanto a consola como a ficheros.
- Podremos tratar con estructuras de control y flujo para el desarrollo de Shell Scripts mediante programación estructurada.
- Sabremos programar estructuras, funciones y paso de parámetros.
- Distinguiremos entre variables locales y globales.
- Seremos capaces de detallar su funcionamiento en el uso de variables en funciones.

5.1. Comandos básicos

Nos referimos como comando en los sistemas de Linux a cualquier instrucción programa susceptible de ser ejecutado, en los siguientes apartados vamos a ir viendo los más básicos.

Los comandos en Linux siguen toda una misma sintaxis básica:

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

Para cada comando, las opciones y argumentos serán diferente, y además habrá muchas veces que estos sean opcionales y no obligatorios. Estas opciones pueden ser cortas, de hecho, puede que solo sean una letra o puede que se formen por una palabra entera. Aunque hay varios tipos de opciones que pueden usarse en un mismo comando, las cortas a veces hasta se pueden unificar en una única expresión, pero eso dependerá del comando. En Linux, comandos y opciones son sensibles a mayúsculas y minúsculas, por lo que hay que escribir siempre de manera correcta el comando o puede que nos de error o que el resultado no sea el deseado.

Listando archivos

El comando para listar archivos en Linux es `ls`.

El nombre de este comando viene derivado de la palabra *LiSt*, porque lista los nombres de los ficheros que se encuentran en un directorio en concreto. Si no le indicamos nada al fichero, será el directorio en el que nos encontremos en ese momento en concreto, pero le podemos pasar como argumentos el directorio que queremos que liste.

```
alumno@Ubuntu:~$ ls -l /home/alumno
total 36
drwxr-xr-x 3 alumno alumno 4096 jul 28 11:37 Descargas
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Documentos
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Escritorio
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Imágenes
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Música
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Plantillas
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Público
drwx------ 5 alumno alumno 4096 jul 11 09:43 snap
drwxr-xr-x 2 alumno alumno 4096 jul 11 09:09 Vídeos
```

Imagen 1. ls en directorio */home/alumno*

Desplegando el contenido de un archivo

Uno de los principales comandos que tenemos a la hora de mostrar en la línea de comandos la información de un archivo es:

```
cat [archivos] [ { > | >> | < | << } archivo ]
```

Se muestra la información de todos los archivos concatenada de manera secuencial. Además, la salida puede ser redireccionada a un archivo con las siguientes opciones:

- sobrescribe en caso de que el archivo exista, si no es así, lo crea.
- **>>** el contenido se añade a un fichero existente justo después del que ya almacena. Si no existe el fichero lo crea.

Esta orden entonces podemos decir que se usa para crear archivos de una extensión corta rápidamente.

Imagen 2. Comando cat

Desplegando texto

El comando `echo` se usa para mostrar por pantalla los argumentos que se le pasen al comando. La salida también se podría redirigir si quisiéramos al igual que con el comando anterior.

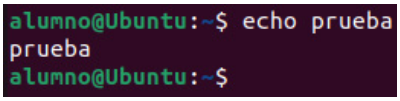

Imagen 3. Comando echo

Copiando archivos

Si queremos copiar archivos debemos de usar el comando `cp`. La sintaxis de este comando es:

```
cp [-irR] archivos_que_copiar destino
```

Si en el destino tenemos archivos de igual nombre, por defecto se sobrescribirá su información al copiar los nuevos. Las opciones más usadas por este comando son:

- `i:` pedirá confirmación para la copia si se tiene que sobrescribir un archivo.
- `r o R:` se copia de manera recursiva cuando se usen directorios.

En el siguiente ejemplo vemos como se copia el directorio *dirprueba* dentro de *Descargas:*

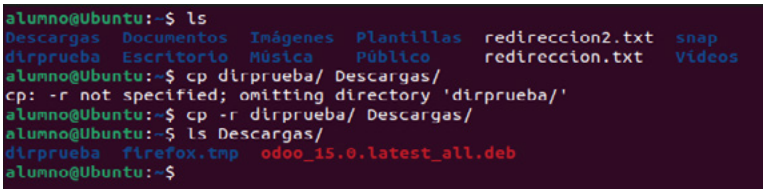

Imagen 4. Comando cp

Hemos visto en el ejemplo, que si no se usa la opción `-r`, no nos dejará copiar el directorio.

PARA TENER EN CUENTA...

Debemos de distinguir que el copiar archivos o directorios no los elimina del origen, sino que simplemente plasma una copia.

Desplegando los procesos del sistema

Los procesos se identifican gracias a un identificador único denominado *PID, Identificador de proceso*. El *PCB* de cada proceso almacena información acerca de este, principalmente:

- El PID del proceso.
- Identificación del proceso padre, PPID.
- Usuario propietario.
- Valores del estado del proceso en el momento de producirse el cambio de contexto.
- Estado.
- Valores de referencia de memoria RAM.
- Ficheros abiertos.
- Buffers de memoria usados.

Para obtener información acerca de los procesos del sistema usaremos el comando `ps [modificadores]`. Este comando tiene un sinfín de opciones y una potencia mayor aún, por lo que para poder verlos todos debemos de acceder a su manual de ayuda con el comando `man ps`.

No obstante, los principales modificadores son:

- Para obtener información de todos los procesos del sistema:
  - `ps aux`
  - `ps -ef`
- Para imprimir información jun a un árbol de procesos.
  - `ps axjf`

```
alumno@Ubuntu:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.6 168160 13384 ?        Ss   08:02   0:01 /sbin/init s
root           2  0.0  0.0      0     0 ?        S    08:02   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   08:02   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   08:02   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   08:02   0:00 [netns]
root           6  0.0  0.0      0     0 ?        I    08:02   0:00 [kworker/0:0
root           7  0.0  0.0      0     0 ?        I<   08:02   0:00 [kworker/0:0
root          10  0.0  0.0      0     0 ?        I<   08:02   0:00 [mm_percpu_w
root          11  0.0  0.0      0     0 ?        S    08:02   0:00 [rcu_tasks_r
root          12  0.0  0.0      0     0 ?        S    08:02   0:00 [rcu_tasks_t
root          13  0.0  0.0      0     0 ?        S    08:02   0:00 [ksoftirqd/0
root          14  0.0  0.0      0     0 ?        I    08:02   0:00 [rcu_sched]
root          15  0.0  0.0      0     0 ?        S    08:02   0:00 [migration/0
root          16  0.0  0.0      0     0 ?        S    08:02   0:00 [idle_inject
root          17  0.0  0.0      0     0 ?        S    08:02   0:00 [cpuhp/0]
root          18  0.0  0.0      0     0 ?        S    08:02   0:00 [kdevtmpfs]
root          19  0.0  0.0      0     0 ?        I<   08:02   0:00 [inet_frag_w
root          20  0.0  0.0      0     0 ?        S    08:02   0:00 [kauditd]
root          21  0.0  0.0      0     0 ?        S    08:02   0:00 [khungtaskd]
root          22  0.0  0.0      0     0 ?        S    08:02   0:00 [oom_reaper]
root          23  0.0  0.0      0     0 ?        I<   08:02   0:00 [writeback]
root          24  0.0  0.0      0     0 ?        S    08:02   0:00 [kcompactd0]
root          25  0.0  0.0      0     0 ?        SN   08:02   0:00 [ksmd]
root          26  0.0  0.0      0     0 ?        SN   08:02   0:00 [khugepaged]
root          72  0.0  0.0      0     0 ?        I<   08:02   0:00 [kintegrityd
root          73  0.0  0.0      0     0 ?        I<   08:02   0:00 [kblockd]
root          74  0.0  0.0      0     0 ?        I<   08:02   0:00 [blkcg_punt_
```

Imagen 5. Comando ps aux

El comando `ps aux` muestra una serie de información que se establece en la cabecera:

- Usuario propietario del proceso.
- PID del proceso.
- CPU consumida en porcentaje.
- Memoria RAM consumida en porcentaje.
- Tamaño del proceso en la memoria virtual en KB.
- Tamaño de la memoria residente de proceso en KB.
- Terminal de lanzamiento.
- Estado del proceso.
- Tiempo de inicio del proceso.
- Tiempo de CPU consumido.
- Comando que lo ejecuta.

Los estados de un proceso en el comando pueden ser los siguientes:

Estados de un proceso	
Estado	**Descripción**
R	Ejecutándose o listo para ser ejecutado. (*Runnable*)
S	Bloqueado o durmiendo (*Sleeping*).
T	Parado (*Trace*).
Z	Zombi (proceso muerto pero el proceso padre no ha detectado su final).
I	Inactivo en creación (*idle*)
N	Con prioridad menor de lo normal (*NICE*).
<	Con prioridad mayor de lo normal.
+	Se encuentra en el grupo de procesos en primer plano.
s	Proceso líder de sesión.
L	Proceso multihilo.

Si en cambio, lo que queremos es ver una actualización constante de cómo evolucionan los procesos del sistema, el comando a ejecutar será `top`, y para salir de este comando pulsaremos la tecla "*q*".

Si ejecutamos el comando veremos que antes de listar los procesos, tenemos una serie de líneas que nos aportan información sobre el sistema.

- Línea 1: hora actual, tiempo del sistema encendido, número de usuarios y carga media en intervalos de 1,5 y 15 minutos, respectivamente.
- Línea 2: número de tareas, número de proceso en estad, ejecutándose o listos, bloqueados o hibernando parados y zombis respectivamente.
- Línea 3: tiempos de CPU, de usuario, *kernel*, etc.
- Línea 4: tamaño en MB de memoria física en total, libre. Usada y utilizada por buffer.
- Línea 5: tamaño en MB de memoria virtual total, libre usada y disponible.

Una vez terminadas estas líneas, la salida de procesos es similar a ps y sus opciones al igual que con el comando anterior, son muchas y muy específicas, por lo que antes d usarlo es recomendable leer su manual.

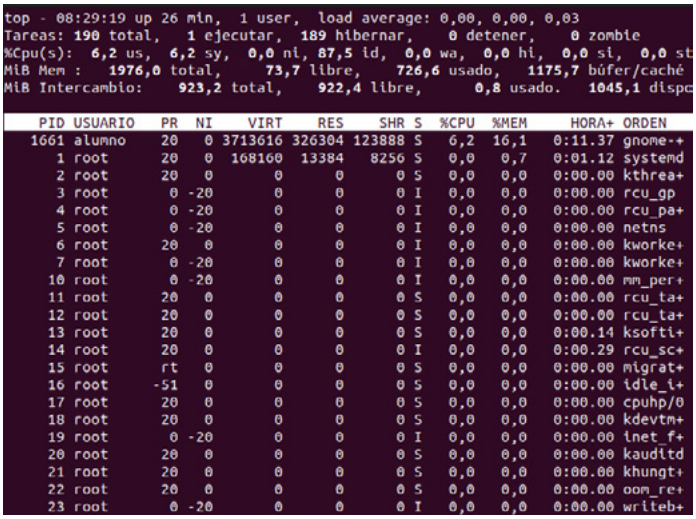

Imagen 6. Comando top.

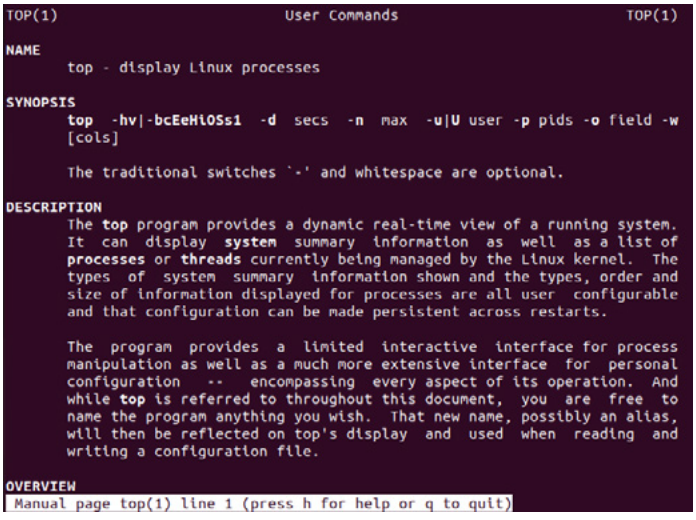

Imagen 7. man top.

Resumen de comandos

Comandos básicos de Linux	
Comando	**Descripción**
`apropos`	Informa sobre un comando
`cat`	Visualiza el contenido de un fichero
`cd`	Cambia de directorio
`cmp, diff`	Compara ficheros
`cp`	Copia archivos
`chgrp`	Cambia el grupo de un fichero
`chmod`	Se cambian los permisos de un fichero o directorio
`chown`	Cambia el usuario propietario de un fichero o directorio
`file`	Indica el tipo de un fichero
`find`	Busca ficheros o directorios
`grep`	Busca un patrón en un fichero o salida de un comando
`head`	Visualiza x líneas del comienzo de un fichero
`info`	Similar a `man`
`less`	Similar a **cat**
`locate`	Busca ficheros
`ls`	Lista el contenido de un fichero
`man`	Nos enseña el manual sobre un comando
`mkdir`	Crea un directorio
`more`	Igual a **cat**
`mv`	Mueve ficheros
`pwd`	Muestra el directorio en que nos encontramos
`rm`	Borra ficheros
`rmdir`	Borra directorios
`tail`	Visualiza x líneas del final del fichero
`umask`	Cambia los permisos por defecto de los ficheros que se crean
`wc`	Cuenta líneas/palabras/caracteres
`whatis`	Breve referencia sobre un comando
`whereis`	Busca ficheros binarios

5.2. Entrada y salida

En cuanto a la programación en *Shell Script* nos referimos, lo primero que vamos a enunciar es como se deben de estructuras los ficheros creados y como los debemos de guardar.

Los ficheros de *Shell Script* se guardan con la extensión *.sh* y todos deben de empezar por la siguiente línea:

```
#!/bin/bash
```

Luego, para poder ejecutar estos ficheros como programas, lo más recomendable será usar la siguiente estructura:

```
./script.sh
```

5.2.1. Entrada y salida por consola

Cuando queremos crear un primer *script*, lo primero que debemos saber es que la sentencia *echo*, nos muestra un texto por pantalla. Su sintaxis es la siguiente:

```
echo opciones “Texto a mostrar”
```

Además, podemos añadirle la opción *-n* para que agregue un salto de línea.

La otra sentencia introductoria es *read*, que se usa para la entrada de datos mediante el teclado para nuestro programa. Su estructura es la siguiente:

```
read variable
```

Las variables realmente se usan para almacenar datos temporales en un espacio de memoria, y se pueden indicar como queramos, lo común es usar las letras x, y, z, i, etc., como en matemáticas.

Cuando queramos trabajar con el contenido de una variable, por ejemplo, para mostrarla, bastará con indicar antes de la variable el símbolo dólar, *'$'*. La sentencia *read* puede leer varios datos al mismo tiempo, y sobrescribir el contenido de una variable.

```
#!/bin/bash
echo -n "Aquí escribimos el texto: "
read x
echo "Aquí mostramos el resultado: $x"
echo -n "Podemos incluso tomar más de un dato: "
read x y
echo "Ahora mostramos los dos datos, primero $x y después $y"
```

Imagen 8. Ejemplo de *script* 1.1

Cuando creemos nuestro primer *script*, por lo general este irá sin los permisos de ejecución, por lo que habrá que otorgárselos.

```
alumno@Ubuntu:~$ chmod u+x prueba1.sh
alumno@Ubuntu:~$
```

Imagen 9. Ejemplo de *script* 1.2

A continuación, vemos como quedaría la salida del *script* de la imagen anterior:

```
alumno@Ubuntu:~$ ./prueba1.sh
Aquí escribimos el texto: Hola
Aquí mostramos el resultado: Hola
Podemos incluso tomar más de un dato: Hola y
Ahora mostramos los dos datos, primero Hola y después y
alumno@Ubuntu:~$
```

Imagen 10. Ejemplo de *script* 1.3

5.2.2. Redirección entrada y salida

Por lo general, la gran mayoría de comandos de una terminal reciben la información que procesar mediante un flujo de entrada y los datos son enviados o mostrados mediante otro flujo de salida. Dependiendo del comando, el SO por defecto asigna entradas y salidas estándar para los dos flujos respectivamente.

Por defecto en el sistema existen tres ficheros que cumplen con entradas y salidas estándar, que son:

- Entrada estándar (stdin).
- Salida estándar (stdout).
- Salida de errores estándar (stderr).

Estos ficheros tienen un número que los describe, 0, 1 y 2, respectivamente.

Suele asociarse la entrada estándar al teclado y la salida estándar y salida de errores estándar a la pantalla, pero esto no es siempre así, porque hay veces en las que las entradas o salidas se pueden nutrir de otros ficheros del sistema.

Por último, como veremos a continuación este flujo de entrada y salida puede cambiarse con diversas opciones de los comandos.

Redirecciones de las salidas estándar

Como ya vimos cuando hablábamos del comando `cat`, se puede usar el operador `>` para que la salida de un comando se redirija y se añada a un fichero en vez de mostrarse por pantalla.

Se sigue la siguiente sintaxis:

```
Comando > fichero
```

Como vimos, si el fichero no existe, lo crea. Y si existe, sobrescribe su contenido. Podríamos además usar la expresión `1>` que daría el mismo resultado al llevar el descriptor del fichero de salida estándar.

Se puede usar también el operador `>>`, que hará la misma función, pero en este caso si el fichero existe, el contenido se anexa al que ya tiene, pero no se sobrescribe, si no existe, lo crea. La sintaxis es la misma que la anterior.

Redirecciones de la entrada estándar

Al igual que se redirecciona la salida, también se puede redireccionar la entrada, que se ejecuta con el operador `<`.

La sintaxis que sigue es:

```
Comando < fichero
```

También se puede usar un tipo de redirección, que lo que hace es que permite escribir tantas líneas como queramos hasta que se encuentre con el delimitador que nosotros establezcamos.

La sintaxis de este es como la siguiente:

```
cat <<delimitador
```

En la siguiente imagen podemos ver un ejemplo de cada uno de los dos casos:

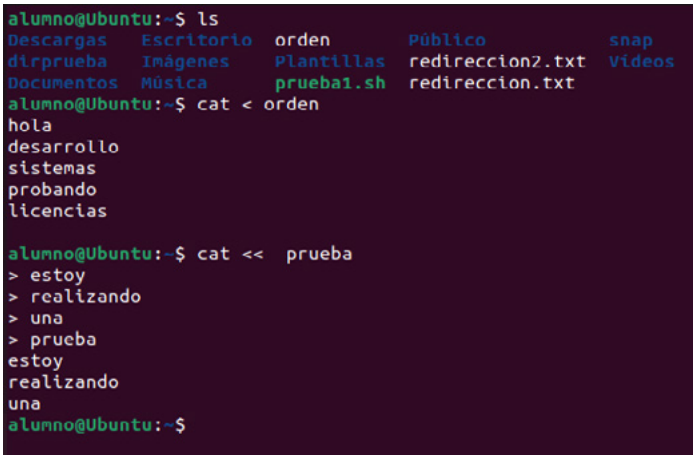

Imagen 11. Redirecciones 1

Como podemos ver en la imagen anterior, al usar la redirección con un delimitador, cuando escribimos en el *prompt* dicho delimitador, la orden muestra el resultado y termina.

Redirección de la salida de error estándar

Habrá ocasiones en las que un comando nos de error y esto es normal, porque nos podemos equivocar en la sintaxis, en los argumentos, etc. Cuando esto sucede, se suele mostrar un resultado de error por pantalla, pero si por ejemplo queremos almacenar estos errores para llevar un registro, podríamos hacerlo también, usando el operador `2>`. Su sintaxis es la que sigue:

```
comando 2> fichero
```

Se podría también usar el operador `2>>` para que el fichero no se sobrescriba como pasaba con la salida estándar.

Por ejemplo, si realizamos un `ping` a una dirección que no existe, se vería del siguiente modo:

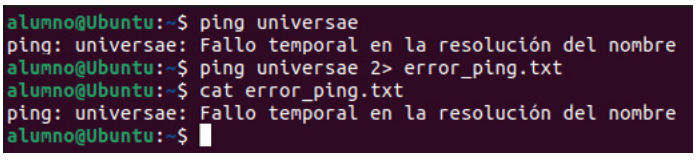

Imagen 12. Redirecciones 2

Redirección de la salida estándar y la salida de error estándar al mismo destino

Al igual que se redirigen la salida estándar y la salida de errores estándar, también podemos realizar una redirección de ambas a la vez al mismo fichero siguiendo las sintaxis anteriores, pero usando el operador `&>`.

Si seguimos con el ejemplo del `ping` anterior:

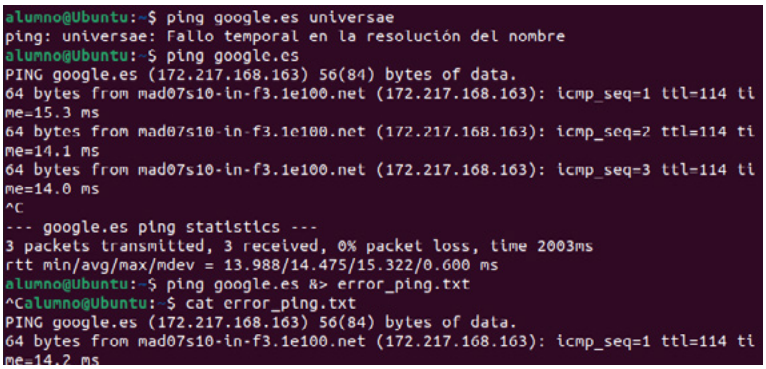

Imagen 13. Redirecciones 3

Podemos fijarnos en que el ping se ha parado de manera natural porque era continuo para que mostrase algún tipo de error

Redirección de la salida estándar y la salida de error estándar de una orden con la entrada estándar de otra orden

Tenemos también en Linux la opción de concatenar la salida de varios comandos para que sirvan como entrada a otros y viceversa. Esto se realiza con el uso de *tuberías o pipes* que se representan con el símbolo ASCII "|". La sintaxis de estas redirecciones es muy sencilla:

```
Comando1 | comando2
```

Para redireccionar ambas salidas, estándar y de error estándar se usa el símbolo `|&` siguiendo la misma sintaxis que la salida estándar.

Y, además, entre comandos podemos usar el comando `tee` para añadir información a la salida del anterior y consecuentemente a la entrada del siguiente.

Vamos a ver un ejemplo:

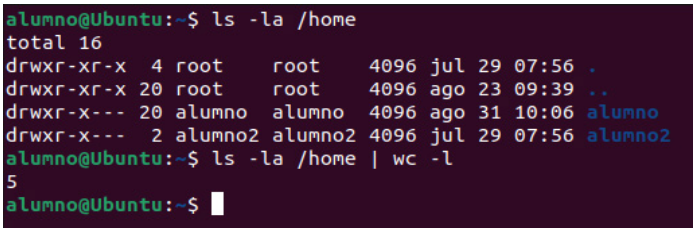

Imagen 14. Redirecciones 4

Podemos ver que la salida del primer comando mostraba 4 líneas, por eso cuando hemos ejecutado la concatenación siguiente, nos ha devuelto dicho número.

Redirección de la salida de error a la salida estándar

Podemos usar el operador `2>&1` si queremos que la salida de error salga por el mismo sitio que la salida estándar, pero realmente no tiene mucho efecto y no suele usarse.

5.3. Expresiones

Hay una serie de comandos que nos permiten usar las expresiones regulares para poder buscar o filtrar por patrones específicos.

Las expresiones regulares se componen de caracteres combinados con operadores, y en el siguiente cuadro podemos ver las principales:

Expresiones regulares	
Expresión	**Descripción**
^	Principio de línea
$	Final de línea
\<	Principio de palabra
\>	Final de palabra
.	Cualquier carácter simple excepto el salto de línea
[]	Conjunto de caracteres
[^]	Cualquier carácter no contenido
[-]	Rango
*	Cero o más ocurrencias del elemento que lo precede
+	Uno o más ocurrencias del elemento que lo precede
?	El elemento precedente es opcional
()	Agrupación
|	O uno u otro
{n}	El elemento precedente se repite n veces
{n,}	El elemento precedente se repite al menos n veces
{n,m}	El elemento precedente se repite un mínimo de n veces y un máximo de m veces

5.3.1. Comando grep

La sintaxis del comando `grep` para filtrar es:

```
grep [-nvlicw] patrón ficheros
```

Y sus opciones son las siguientes:

- `l`: solo se van a mostrar los ficheros que contengan el patrón que hemos seleccionado.
- `i`: no distingue entre mayúsculas y minúsculas.
- `c`: nos muestra el número total de las líneas que contienen el patrón que se ha especificado.
- `w`: busca el patrón en una palabra completa y no como parte de una cadena de caracteres.
- `n`: se muestra el número de línea de cada una de las que tengan el patrón especificado.
- `v`: busca las líneas que no contengan el patrón que se ha pasado.

Por ejemplo, vamos a filtrar por el usuario *root* en */etc/group*:

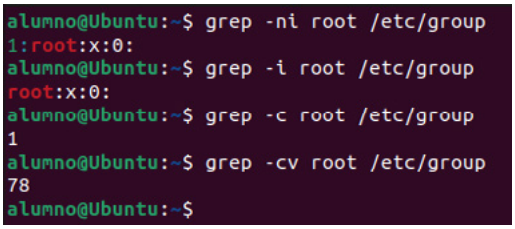

Imagen 15. Comando grep

Podemos ver las cuatro ocasiones que lo hemos usado y que su resultado ha sido correcto.

Existe la opción `-E` o el comando `egrep` que permite el uso de las expresiones regulares extendidas para patrones más complejos.

También existe la opción `--color` para marcar de color los resultados de coincidencia del patrón.

5.3.2. Comando sed

Las expresiones regulares también se pueden usar mediante el comando `sed`. Aunque su función puede ser la de búsqueda, a la hora de la verdad el uso de este comando viene más enfocado al remplazo de texto. Su sintaxis es la siguiente:

```
sed -n[r] '/REGEX/p' FICHERO
```

o

```
Comando | Sed -n[r] '/REGEX/p'
```

5.4. Control de flujo

Los controles de flujo en los distintos lenguajes de programación los conocemos como estructuras de control. Cada una de estas sentencias se encuentran delimitadas por una palabra reservada o un operador de control al principio y otro al final, que se suele usar como un *terminador*.

Las condiciones son una de las principales estructuras en la programación.

5.4.1. Condición if – then – else – if

En el *Shell Script* de Linux podemos establecer condiciones para realizar comparaciones y así establecer un primer filtro de resultados.

La sentencia que se usa para las condiciones por lo general es *if*, que tiene la siguiente estructura:

```
if [ condición ]; then
        Sentencia1
        Sentencia2
        …
        Sentencia n
else
        Sentencia1
        Sentencia2
        …
        Sentencia n
fi
```

Podemos ver que esta condición se cierra con la misma sentencia inversa, es decir, *fi*, pero entre medias, tenemos *else*, que nos indica que debemos de hacer si no se cumple la condición anterior a *then*.

Para las condiciones de *if*, lo normal es que trabajemos con números, para lo que tenemos las siguientes condiciones principales:

Operadores de comparación	
Condición	**Significado**
-gt	Mayor que
-ge	Mayor o igual que
-eq	Igual que
-ne	No igual que
-le	Menor o igual que
-lt	Menor que

Si lo que queremos es comparar caracteres exactos, habrá que usar los numeradores siguientes: >, >=, ==, <>, <=, <; todos son respectivos a los del cuadro anterior. En este caso habrá que poner el carácter a comparar entre comillas "".

Ahora que sabemos esto, un ejemplo de condición con *if*, podría ser el siguiente.

```
#!/bin/bash
echo -n "Introduce un número: "
read x
if [ $x -gt 0 ]; then
        echo "El número $x es mayor que 0"
else
        echo "El número $x es menor que 0"
fi
```

Imagen 16. Condiciones *if* 1

Su resultado sería…

```
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: 1
El número 1 es mayor que 0
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: -1
El número -1 es menor que 0
alumno@Ubuntu:~$
```

Imagen 17. Condiciones *if* 2

Otra característica de estas condiciones es que podemos anidar un *if* dentro de otro, eso sí, finalizando siempre cada uno cuando le toque.

```
#!/bin/bash
echo -n "Introduce un número: "
read x
if [ $x -ne 0 ]; then
        if [ $x -gt 0 ]; then
                echo "El número $x es mayor que 0"
        else
                echo "El número $x es menor que 0"
        fi
else
        echo "El número es igual a 0"
fi
```

Imagen 18. Condiciones *if* 3

```
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: -2
El número -2 es menor que 0
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: 0
El número es igual a 0
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: 2
El número 2 es mayor que 0
alumno@Ubuntu:~$
```

Imagen 19. Condiciones *if* 4

Por último, en estos condicionales, tenemos una sentencia anidada que sirve para cuando dentro de *else*, queremos añadir otra condición, la sentencia es *elif* (acortamiento de *else if*) y se añade la condición como con *if* pero sin necesidad de terminar con *fi*.

```
#!/bin/bash
echo -n "Introduce un número: "
read x
if [ $x -gt 0 ]; then
        echo "El número $x es mayor que 0"
elif [ $x -eq 0 ]; then
        echo "El número es igual a 0"
else
        echo "El número $x es menor que 0"
fi
```

Imagen 20. Condiciones *if* 5

```
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: 2
El número 2 es mayor que 0
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: 0
El número es igual a 0
alumno@Ubuntu:~$ ./prueba2.sh
Introduce un número: -2
El número -2 es menor que 0
alumno@Ubuntu:~$
```

Imagen 21. Condiciones *if* 6

Hay que añadir, por último, que tanto los *else*, como los *elif*, son sentencias opcionales dentro de *if* y, además, no pueden funcionar sin esta primera sentencia.

5.4.2. Condición *case*

Como hemos enunciado antes, lo principal en cuanto a condiciones suele ser la sentencia *if*, pero dependiendo de que condición queramos implantar, en vez de tener varios *if* anidados, podemos usar la condición *case*.

Esta condición se basa en distintos casos que según sean o no coincidentes con el valor pasado, nos mostrará la sentencia que deseemos, siguiendo la siguiente estructura:

```
case condición in
        Case1)
                Sentencias
        ;;
        Case2)
                Sentencias
        ;;
        Case n)
                Sentencias
        ;;
        *)
                Sentencias
esac
```

Un ejemplo de una condición con *case* puede ser el siguiente:

```
#!/bin/bash
echo -n "Indica un número: "
read x
case $x in
        1)
                echo "Has escrito el $x"
        ;;
        2)
                echo "Has escrito el $x"
        ;;
        3)
                echo "Has escrito el $x"
        ;;
        4)
                echo "Has escrito el $x"
        ;;
        5)
                echo "Has escrito el $x"
        ;;
        *)
                echo "Has escrito un número mayor que 5"
        ;;
esac
```

Imagen 22. Condiciones *case* 1

Si lo probamos, vemos que funciona a la perfección.

```
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 1
Has escrito el 1
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 2
Has escrito el 2
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 3
Has escrito el 3
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 4
Has escrito el 4
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 5
Has escrito el 5
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 6
Has escrito un número mayor que 5
alumno@Ubuntu:~$ ./prueba3.sh
Indica un número: 10
Has escrito un número mayor que 5
alumno@Ubuntu:~$
```

Imagen 23. Condiciones *case* 2

Se pude observar que en la condición podemos simplemente nombrar la variable.

5.4.3. Bucle for

Los bucles se usan como estructuras de control que se repiten durante un periodo de tiempo marcado por unas condiciones claras.

El primero de los bucles que vamos a ver es el bucle *for*. Este bucle repetirá una serie de sentencias para cada elemento de una lista hasta que esta termine. Su estructura es la siguiente:

```
for variable in elemento1 elemento2 … elementon
do
        Sentencias
done
```

Por ejemplo, podemos hacer un ejemplo con esta estructura como el siguiente:

```
#!/bin/bash
x=1
for i in lunes martes miercolés jueves viernes sábado domingo
do
        echo "El día $x de la semana es $i"
        x=$(($x+1))
done
```

Imagen 24. Bucle *for* 1

```
alumno@Ubuntu:~$ ./prueba4.sh
El día 1 de la semana es lunes
El día 2 de la semana es martes
El día 3 de la semana es miercolés
El día 4 de la semana es jueves
El día 5 de la semana es viernes
El día 6 de la semana es sábado
El día 7 de la semana es domingo
alumno@Ubuntu:~$
```

Imagen 25. Bucle *for* 2

Este tipo de bucles, también se pueden mostrar del siguiente modo:

```
for ((variable=valor; variable condición valor; incremento/decremento)); do
        Sentencias
done
```

Este es el llamado bucle *for* del lenguaje de programación *C*, que se basa en el incremento de una variable.

```
#!/bin/bash
echo "Vamos a contar del 1 al 10"
for ((i=1; i<=10; i++)); do
        echo $i
        if [ $i -lt 10 ]; then
                echo "Ahora se suma un número"
        fi
done
echo "Hemos llegado a 10"
```

Imagen 26. Bucle *for* 3

Vemos que dentro de esto podemos añadir sentencias diferentes y que todo funciona correctamente, como se demuestra en la siguiente imagen:

```
alumno@Ubuntu:~$ ./prueba4.sh
Vamos a contar del 1 al 10
1
Ahora se suma un número
2
Ahora se suma un número
3
Ahora se suma un número
4
Ahora se suma un número
5
Ahora se suma un número
6
Ahora se suma un número
7
Ahora se suma un número
8
Ahora se suma un número
9
Ahora se suma un número
10
Hemos llegado a 10
alumno@Ubuntu:~$
```

Imagen 27. Bucle *for* 4

5.4.4. Bucle while

El siguiente bucle que vamos a ver es el *while*. Este bucle se caracteriza por repetir sentencias siempre que la condición marcada sea verdadera y su estructura es:

```
while [ condición ]
do
        Sentencias
done
```

Un ejemplo de esto puede ser el que vemos a continuación:

```
#!/bin/bash
x=1
while [ $x -lt 10 ]
do
        echo "El número $x sigue siendo menor de 10"
        x=$(($x+1))
done
```

Imagen 28. Bucle *while* 1

Vemos que hemos inicializado la variable primeramente y que, además, después, vamos a incorporar una operación para incrementar la variable y así, poder terminar un bucle, si no tuviéramos esto, se crearía un bucle infinito.

```
alumno@Ubuntu:~$ ./prueba5.sh
El número 1 sigue siendo menor de 10
El número 2 sigue siendo menor de 10
El número 3 sigue siendo menor de 10
El número 4 sigue siendo menor de 10
El número 5 sigue siendo menor de 10
El número 6 sigue siendo menor de 10
El número 7 sigue siendo menor de 10
El número 8 sigue siendo menor de 10
El número 9 sigue siendo menor de 10
alumno@Ubuntu:~$
```

Imagen 29. Bucle *while* 2

5.4.5. Bucle until

El bucle *until* hará que las instrucciones se repitan hasta que la condición sea verdadera.

Su estructura es la siguiente:

```
until [ condición ]
do
        Sentencias
done
```

Podemos hacer el ejemplo anterior pero a la inversa.

```
#!/bin/bash
x=1
until [ $x -eq 10 ]
do
        echo "El número $x es menor que 10"
        x=$(($x+1))
done
```

Imagen 30. Bucle *while* 1

```
alumno@Ubuntu:~$ ./prueba6.sh
El número 1 es menor que 10
El número 2 es menor que 10
El número 3 es menor que 10
El número 4 es menor que 10
El número 5 es menor que 10
El número 6 es menor que 10
El número 7 es menor que 10
El número 8 es menor que 10
El número 9 es menor que 10
alumno@Ubuntu:~$
```

Imagen 31. Bucle *while* 2

Si nos fijamos en las imágenes, este sacaría lo mismo que con *while* usando menos sentencias, lo que lo haría más eficiente y, por lo tanto, mejor opción.

5.5. Funciones

En *Shell script* existen una serie de sentencias que se llaman funciones. Estas funciones se pueden insertar en cualquier parte del código de nuestro *script* y pueden ser llamadas al escribir el nombre de la función. Cuando llamamos a una función, lo que hacemos es ejecutar el bloque de código que va dentro de esta.

La estructura de una función es:

```
función () {
        Sentencias
}
```

Tenemos varias aclaraciones que debemos de hacer con respecto a las funciones:

- Lo que se comprende entre las llaves, *{}*, es lo que llamamos el cuerpo de la función.
- Como hemos dicho antes, si queremos llamar a una función para que se ejecute, lo hacemos escribiendo el nombre de la función.
- Siempre hay que definir la función antes de llamarla, si no, dará fallo.
- Debemos intentar llamar a la función con un nombre que nos de una pequeña intuición de que va a realizar la función.

Un ejemplo de función puede ser como lo que a continuación se nos presenta:

```
#!/bin/bash
funcion_prueba () {
        echo "Solo estamos probando"
        echo "Por ejemplo, 1+1 son: " $((1+1))
}

funcion_prueba
```

Imagen 32. Funciones 1

Observemos que si indicamos *$((sentencia)*, se nos devuelve el resultado de la sentencia.

Su salida sería la siguiente:

```
alumno@Ubuntu:~$ ./prueba7.sh
Solo estamos probando
Por ejemplo, 1+1 son:  2
alumno@Ubuntu:~$
```

Imagen 33. Funciones 2

Las mismas funciones también se pueden escribir del siguiente modo:

```
#!/bin/bash
function funcion_prueba () {
        echo "Solo estamos probando"
        echo "Por ejemplo, 1+1 son: " $((1+1))
}

funcion_prueba
```

Imagen 34. Funciones 3

5.5.1. Valores de retorno

En las funciones de *Shell Script* no tenemos un retorno como tal, aunque los llamemos. Con retorno, nos referimos al indicador del resultado de la función, por ejemplo, si todo finaliza con éxito, el valor de retorno de una función será 0. Si ocurre algún error o fallo será valorado con un número del 1 al 255.

Podemos especificar en *Shell Script* el valor de retorno con la palabra *return* o asignándolo a la variable *$?*. Siempre que especifiquemos esto, la función se dará por terminada.

5.5.2. Pasar parámetros

Podemos pasar argumentos o parámetros a un *script* simplemente escribiéndolos seguidos a su ejecución. Además, podemos también asignarle esos parámetros a una función y operar con ellos como variables locales.

La estructura sería la siguiente:

- Para un *script*:

  ```
  ./script.sh parámetro1 parámetro2 … parámetron
  ```

- Para una función sería igual, pero a la hora de llamar a dicha función.

Vamos a tener en cuenta ahora varias consideraciones acerca de los parámetros:

- Los parámetros que pasamos son *$1,$2,…,$n* correlativamente a la posición en la que se introducen.
- La variable *$0* nos indica el nombre del *script*.
- La variable *$#* nos indica cuantos parámetros se han pasado.
- La variable *$** muestra todos los parámetros seguidos. También se puede usar *$@*.

Aquí tenemos un ejemplo práctico de parámetros:

```
#!/bin/bash

funcion_parametros () {
        echo "Mostramos el parámetro 1: $1"
        echo "Mostramos el parámetro 2: $2"
        echo "Mostramos el nombre del script: $0"
        echo "Mostramos el número de parámetros: $#"
        echo "Podemos mostrar todos los parámetros seguidos: $*"
}

funcion_parametros $1 $2
```

Imagen 35. Parámetros 1

```
alumno@Ubuntu:~$ ./prueba8.sh parametro1 parametro2
Mostramos el parámetro 1: parametro1
Mostramos el parámetro 2: parametro2
Mostramos el nombre del script: ./prueba8.sh
Mostramos el número de parámetros: 2
Podemos mostrar todos los parámetros seguidos: parametro1 parametro2
alumno@Ubuntu:~$
```

Imagen 36. Parámetros 2

www.universae.com