# Linux (Unix ) para usuarios


## Pedro Pablo fabrega@jazzfree.com



Esta guía pretende que el lector se familiarice con el trabajo en la shell de un sistema Unix, Linux en particular. Todo el software descrito en la presente guía es software libre, por lo que el lector no debería tener problema en conseguirlo por los cauces habituales, es decir distribuciones de Linux e Internet. Por otro lado esta guía se limita a la descripción de tareas a nivel de usuario, o sea, que todo lo que aquí se indica (o casi todo) se puede hacer sin tener acceso a la cuenta de root. Las tareas de administración serán otro capítulo.


## Tabla de contenidos


**Introducción al Unix** ..... **3**
**Órdenes para ficheros y directorios** ..... **10**
**Comprimir, descomprimir y agrupar ficheros** ..... **19**
**Otras órdenes de usuario** ..... **22**
**Gestión de procesos** ..... **32**
**Entrada y salida** ..... **38**
**Tratamiento de ficheros de texto** ..... **42**
**Permisos y propietarios** ..... **46**
**Shell** ..... **52**
**Ejecución y agrupación de órdenes** ..... **58**
**Programas de shell** ..... **62**
**Claves** ..... **85**

# Introducción al Unix

Vamos a describir las generalidades de Unix de una forma ligera. Realizaremos una descripción de los elementos que son de utilidad

## Historia de Unix

El sistema operativo Unix tiene su origen en los laboratorios Bell de AT&T en los años 60. Estos laboratorios trabajaban en un sistema operativo nuevo llamado MULTICS (Multiplexed Information and Computing System. Este proyecto fue un fracaso, pero los componentes del equipo adquirieron una gran experiencia durante su desarrollo.

Uno de los componentes del equipo, Kem Thompson, escribió un juego llamado "Space Travel" y escribió un sistema operativo para poder jugar con él. Consiguió que dos personas pudieran jugar simultáneamente, con este sistema operativo, que por un juego de palabras en comparación con MULTICS, lo llamó UNICS.

Este sistema UNIX estaba escrito en ensamblador, lo que dificultaba que se pudiera usar en máquinas con distintos procesadores. Viendo el problema, Ken Thomson y Denis Ritchie crearon un lenguaje de programación de alto nivel, el lenguaje C, en el cual reescrbieron todo el sistema operativo lo que permitió que se pudiera usar en prácticamente cualquier tipo de ordenador de la época. Sólo las partes críticas seguían en ensamblador.

Más tarde un decisión judicial obligó a AT&T a dejar de vender su sistema operativo. Esta compañía dejó las fuentes del sistema operativo a diversas universidades, las cuales, junto con otras empresas, continuaron el desarrollo del sistema operativo Unix e hizo que tuviera una enorme difusión.

## Versiones de Unix

Existen muchas versiones de Unix. Diversas empresas han desarrollado sus propios sistemas operativos basados en Unix. Podemos citar AIX, HPUX, Solaris, SunOS, IRIX, Xenix, SCO, FreeBSD, Linux.

## El sistema de ficheros

El sistema de ficheros es la organización lógica del disco que nos permite almacenar la información en forma de ficheros de un modo totalmente transparente. Esta palabra tan utilizada significa que no tenemos que preocuparnos de pistas, sectores, cilindros y otras menudencias.

Cada partición del disco, o cada disquete debe tener un sistema de ficheros si queremos almacenar información en forma de fichero.

Tipos de sistemas de ficheros

Cada sistema operativo posee su propia organización lógica del disco para poder almacenar la información, y la usará normalmente, pero además puede tener la posibilidad de usar particiones propias de de otros sistemas. Entre los tipos de sistemas de ficheros podemos citar:

- ext2: linux nativo. Es el sistema de ficheros que utiliza linux por defecto. Soporta características avanzadas: propietarios, permisos, enlaces, etc.
- ext3: linux nativo con journaling. Similar a ext2 pero con transacciones para evitar que apagados accidentales puedan deteriorar el sistema de ficheros.
- msdos: es la organización clásica de este sistema. Es un sistema de archivos diseñado para un sistema monousuario. Utiliza nombres del tipo 8+3.

- vfat: es una ampliación del sistema de ficheros msdos, con soporte para nombres largos de ficheros. Existen los tipos FAT16 y FAT32, y en ambos casos sólo tienen características monousuario: no admiten propietarios de ficheros y los permisos son muy limitados.
- NTFS: sistema de ficheros de Windows NT. Este sistema de ficheros sí está preparado para utilizarse en entornos multiusuario.
- iso9660: es el sistema de ficheros de los CDs. Este estándar admite ciertas extensiones como «Joliet» o «Rock Ridge» que le añaden ciertas características.

El sistema de ficheros Unix

Los elementos del sistema de archivos son el superbloque, i-nodos y bloques de datos. En el capítulo de administración se verá esto con más detalle.

En primer lugar tenemos el superbloque, que contiene la descripción general del sistema de ficheros: Tamaño, bloques libres, tamaño de la lista de i-nodos, i-nodos libres, verificaciones, etc.

En segundo lugar tenemos los i-nodos. Un i-nodo contiene toda la información sobre cada conjunto de datos en disco, que denominamos fichero:

- Donde se almacenan los datos, es decir lista de bloques de datos en disco. Esto son una serie de punteros o direcciones de bloques que indican bien donde están los datos en disco, o bien donde están los bloques que tienen más direcciones de bloques de datos (bloques indirectos).
- Quien es el propietario de los datos, un número que lo identifica (UID o User Identifier), y a qué grupo pertenece el fichero GID Group Identifier).
- Tipo de fichero: regular, es decir un fichero que contiene información habitual, datos o programas; dispositivo, un elemento destinado a intercambiar datos con un periférico, enlace, un fichero que apunta a otro fichero; pipe, un fichero que se utiliza para intercambiar información entre procesos a nivel de núcleo. directorio, si el elemento no contiene datos sino referencias a otros ficheros y directorios.
- Permisos del fichero (quien puede leer(r), escribir(w) o ejecutar(x)). Estos permisos se asignan a se asignan de forma diferenciada a tres elementos: el propietario, el grupo (indicados con anterioridad) y al resto de los usuarios del sistema.
- Tamaño del fichero.
- Número de enlaces del fichero. Es decir cuantos nombres distintos tiene este fichero Hay que observar como el nombre de un fichero no forma parte del i-nodo. El nombre de fichero se asocia a un i-nodo dentro de un fichero especial denominado directorio. Esto le proporciona al sistema de ficheros la posibilidad de que un mismo i-nodo pueda tener varios nombres si aparece en varios directorios o con distintos nombres.

## Entrando a un sistema Unix

Unix es un sistema multiusuario real, es decir, pueden haber varias personas trabajando a la vez en distintas terminales con un mismo host Unix.

## Iniciando una conexión

Existen diferentes métodos para poder conectar los terminales al sistema:

- En primer lugar podemos conectarnos a un sistema Unix a través de el puerto serie (RS232), con una terminal no inteligente o bien con otro equipo y un emulador de terminales. En ambos casos existe un programa que atiende las solicitudes de

conexión a través del puerto serie. Cuando hay una solicitud de conexión, este programa la atiende solicitando al usuario que se identifique ante el sistema. Cuando termina la conexión, este programa se reactiva para seguir atendiendo nuevas solicitudes.

- Mediante tarjeta de red. En este caso, tenemos un programa que escucha las solicitudes de conexión a través de la tarjeta de red. Cuando llega una solicitud este programa se desdobla de forma que una parte atiende la conexión y otra continúa atendiendo nuevas conexiones. Así podemos tener más de una conexión a través de la tarjeta de red.
- La consola. Evidentemente, en un sistema Unix también podemos trabajar desde el teclado y monitor que están conectados directamente al sistema.

## Iniciando una sesión

Una vez que hemos conseguido conectarnos a un sistema Unix tenemos que iniciar una sesión de trabajo. Como dijimos, Unix, y Linux como tal también, son sistemas multiusuario reales, y esto exige que el usuario se presente al sistema y que este lo acepte como usuario reconocido. Así cada vez que iniciamos una sesión Linux nos responde con

```
Login:
```

a lo que nosotros debemos responder con nuestro nombre de usuario. Acto seguido, Linux nos solicita una clave para poder comprobar que somos quien decimos que somos:

```
Password:
```

En este caso tecleamos la clave de acceso. Por motivos de seguridad esta clave no aparecerá en la pantalla. Si la pareja nombre de usuario/clave es correcta el sistema inicia un intérprete de órdenes con el que podemos trabajar. Habitualmente será el símbolo $, aunque puede ser también el símbolo % (si usamos una shell C). Cuando es el administrador (root) quien está trabajando en el sistema, el indicador que aparece es #.

### La base de datos de los usuarios

Hemos visto que para iniciar una sesión de trabajo en un sistema Unix teníamos que suministrar al sistema una pareja de nombre de usuario/clave. Estos datos se almacenan en un fichero llamado*/etc/passwd*. Este fichero contiene una línea por cada usuario del sistema. Cada línea consta de una serie de campos separados por dos puntos (:). Estos campos son, en el orden que aparecen:

Nombre de usuario. Es es nombre con el que no presentamos al sistema, con el que tenemos que responder a *Login:* y por el que nos identifica el sistema.

Clave cifrada. El siguiente campo es la clave de acceso al sistema. Esta clave no se guarda como se introduce, sino que se almacena transformada mediante el algoritmo DES para que nadie pueda averiguarla.

UID. Identificador de usuario. Es el número de usuario que tiene cada cuenta abierta en el sistema. El sistema trabaja de forma interna con el UID, mientras que nosotros trabajamos con el nombre de usuario. Ambos son equivalentes.

GID. Identificador de grupo. Es el número de grupo principal al que pertenece el usuario.

Nombre completo de usuario. Este es un campo meramente informativo, en el que se suele poner el nombre completo del usuario.

Directorio personal. Este campo indica el directorio personal de un usuario, en el cual el usuario puede guardar su información.

shell. El último campo indica un programa que se ejecutará cuando el usuario inicie una sesión de trabajo. Normalmente este campo es una shell que proporciona una línea de órdenes para que el usuario trabaje. Ejemplo:

```
usuario:x%6YkH$Ss:505:705:Cuenta de usuario:/home/usuario:/bin/bash
   ^          ^    ^   ^         ^                 ^          ^
   |          |    |   |         |                 |          |
   |          |    |   |         |                 |          shell
   |          |    |   |         |          directorio personal
   |          |    |   |     Nombre completo del usuario
   |          |    |  Número de grupo (GID)
   |          |   Número de usuario (UID)
   |       Clave cifrada
 Nombre de usuario
```

## La shell

Como dije en el punto anterior, la shell es el intérprete de órdenes de un sistema Unix. No hay que confundir la shell con el sistema operativo. El sistema operativo es el núcleo y la shell es un interfaz que nos proporciona utilidades de trabajo y permite establecer una relación con el núcleo. Hay diversas shells, cada una con sus características. Podemos citar:

- Bourne shell (sh)
- Korn shell (ksh)
- Shell C (csh)
- Bourne again shell (bash)

Metacaracteres de la shell

Existen ciertos caracteres que tienen un significado propio para la shell. Estos caracteres son:

```
* ? [ ] [!] |
> 2> >> 2>> >&
< <<
() & $  \   && ||
```

Si en alguna ocasión nos interesa usar este carácter como literal, es decir, que la shell no lo interprete como carácter especial es necesario que esté precedido (protegido) por el carácter de escape \ (barra invertida).

Entrada y salida estándares y de errores

Un sistema Unix dispone de tres vías o canales para comunicarse con el exterior de forma estándar.

Una de ellas, la entrada estándar, se utiliza para introducir datos en la shell; de forma predeterminada está asociada al teclado.

La salida estándar, se utiliza para mostrar información y de forma predeterminada está asociada al monitor (consola).

Por último existe un canal dedicado a mostrar la salida de errores, que de forma predeterminada está asociado a la salida estándar.

En múltiples ocasiones nos puede interesar redirigir alguna de estas salidas a otro canal; para realizar esto disponemos de los metacaracteres :

< redirige la entrada estándar

> redirige la salida estándar. Si esta redirección es a un fichero, lo crea nuevo.

>> redirige la salida estándar. Si esta redirección es a un fichero, la añade al final.

2> redirige la salida de errores, igual que la redirección de la salida estándar.

Veremos algunos ejemplos en los próximos capítulos.

## Organización del almacenamiento en disco

El almacenamiento en disco se organiza en dos elementos básicos, el fichero y el directorio. Un fichero es un conjunto de datos en disco asociado a un i-nodo. Un directorio es un fichero especial donde se asocia un nombre a un número de i-nodo. Esto produce la sensación de que el directorio contiene ficheros y otros directorios. Esta organización nos permite que podamos asignar varios nombres a un mismo conjunto de datos en disco.

Cuando veamos el capítulo de administración del sistema se estudiará todo esto con más profundidad.

Árbol de directorios

Los sistemas Unix se organizan en un único árbol de directorios. Existe un directorio principal, denominado raíz (/) dentro del cual aparecen el resto de directorios que tenga el sistema.

Esto quiere decir que los distintos discos no aparecen como tales, sino que aparecen como un directorio más y esto lo podemos organizar de acuerdo con nuestros intereses. En el siguiente ejemplo podemos observar una forma de organizar un sistema Linux

```
$ mount
/dev/hdb1 on / type ext2 (rw)
none on /proc type proc (rw)
/dev/hdb3 on /usr type ext2 (rw)
/dev/hdb4 on /opt type ext2 (rw)
/dev/hda5 on /tmp type ext2 (rw)
/dev/hda6 on /usr/src type ext2 (rw)
/dev/hda7 on /usr/local type ext2 (rw)
```

Esto quiere decir que la primera partición (1) del segundo disco duro (hdb) contiene el directorio raíz (/). La tercera partición (3) del segundo disco duro (hdb) aparece en en directorio /usr. La séptima partición (7) del primer disco duro (hda) aparece en el directorio */usr/local*.

## Rutas de acceso a ficheros y directorios

### Nombres de ficheros y directorios

El nombre de un fichero o directorio está formado por letras, números y otros caracteres, salvo el carácter /. Debemos evitar los metacaracteres de la shell en los nombres de ficheros. En caso necesario, si el nombre tiene un metacarácter o espacio en blanco, será necesario proteger el nombre de fichero con comillas (").

Los sistemas unix no identifican ninguna extensión en los nombres de ficheros.

### Plantillas para nombres de ficheros

Hay ocasiones en que nos interesa usar nombres de ficheros que hagan referencia, no a un fichero individual, sino a un conjunto de ficheros. Para estos casos tenemos las plantilla. Una plantilla se forma con caracteres normales y mediante los caracteres:

* equivale a cualquier cadena de caracteres. Por ejemplo ab* equivale a todos los ficheros que empiecen por ab.

? equivale a un carácter individual. Por ejemplo ab? equivale a los nombres de ficheros (o directorios) con tres caracteres y los dos primeros son ab.

### Ruta de un fichero

Para indicar la ubicación en disco de un fichero hay que indicar la lista completa de directorios que contienen al fichero. Es decir, un fichero puede estar dentro de un directorio que a su vez está dentro de otro y así varios. En esta lista, que denominaremos ruta de acceso, cada directorio está separado del siguiente directorio por el signo / , y sin dejar espacios en blanco. Por ejemplo:

```
/usr/bin/wc
```

hace referencia la fichero wc que está contenido en el directorio bin, que a su vez está dentro del directorio usr, que está en el directorio raiz.

Para facilitar las cosas existen ciertos directorios especiales:

/ Es el directorio raíz. El superior de todos.

. Es el directorio actual. En el que nos encontramos en un momento dado. El directorio actual se puede cambiar con una simple orden que se verá con posterioridad.

.. Es el directorio superior al que nos encontramos. El único directorio que no tiene directorio superior es el directorio raíz.

### Ruta absoluta

Una ruta absoluta es aquella que parte del directorio raíz. Las rutas absolutas son válidas en cualquier caso. Ejemplo:

```
/home/usuario/.profile
```

### Ruta relativa

Es una ruta que parte del directorio actual como origen. Esta ruta sólo es válida desde un directorio actual concreto , es decir es relativa a un directorio.

```
../../.profile
```

En este caso estamos haciendo referencia al fichero .profile que está dos directorios por encima del directorio actual.

## Propietarios y Permisos

### Propiedad

Como vimos en la descripción del sistema de ficheros, cada *i-nodo* guarda un espacio para indicar los números de usuario y grupo. Entonces cada *i-nodo* pertenece a un usuario y a un grupo. También el usuario que tenga el UID (número de identificación) descrito en la base de datos de usuario */etc/passwd* que coincida con el del fichero será su propietario. Esto también es válido para directorios.

### Permisos

Cada i-nodo guarda un espacio para almacenar los permisos bajo los cuales se puede acceder a un fichero. Los permisos se aplican al propietario, al grupo y al resto de los usuarios. Unix dispone de tres permisos, lectura(r), escritura(w) y ejecución(x). En total tendremos nueve bits que indican los distintos permiso en el siguiente orden: usuario, grupo, otros. Los permisos los podemos expresar en formato octal. Por ejemplo el valor 751 indicará:

```
usuario 7 = 111 = rwx
grupo   5 = 101 = r-x
otros   1 = 001 = --x
```

De esta forma los permisos quedan rwxr-x--x

Hay que tener en cuenta que los permisos tienen distinto significado si se aplican a un fichero o a un directorio.

Permiso de lectura Permite o evita que alguien pueda leer el contenido de un fichero o de un directorio.

Permiso de escritura En el caso de un fichero, el permiso de escritura permite modificarlo o borrarlo. En el caso de un directorio este permiso da la posibilidad de crear o borrar ficheros de un directorio.

Permiso de ejecución En el caso de un fichero, permite que sea ejecutado por quien tenga el permiso. En el caso de un directorio, el permiso de ejecuación permite entrar en él.

## Órdenes

Son programas que se utilizan para gestionar el sistema y las labores de usuario y administración. Los mayoría de las órdenes siguen un formato estándar:

orden opciones argumentos donde:

- orden: es el nombre de la orden que queremos ejecutar.
- opciones: son modificadores del comportamiento de la orden. Las opciones van precedidas de un guion (-) si son caracteres simples (letras), o por dos guiones (--) si la opción es una palabra completa. Cuando las opciones son caracteres simples se pueden agrupar con un solo guion. (Por este motivo, cuando las opciones son palabras completas van precedidas por dos guiones, para evitar confusiones). Las opciones pueden tener argumentos.
- argumentos: es la información que necesita una orden para poderse ejecutar. Ejemplo:

```
$ ls -la /etc
```

donde

$ Es el indicador de la línea de órdenes del sistema (promtp).

ls Es la orden para mostrar la lista de ficheros de un directorio.

-la aplicamos las opciones l y a que indican como queremos obtener la lista.

/etc El directorio del cual queremos obtener la lista de ficheros. Este sería el argumento de la orden ls.

### Ejecución de una orden

Para ejecutar una orden simplemente lo escribimos en la línea de orden con sus opciones y argumentos y pulsamos enter. En algunos casos puede ser necesario indicar la ruta de búsqueda del fichero que contiene la orden para poderlo ejecutar. (Ver la variable de entorno *PATH*).

También en una misma línea podemos lanzar varias órdenes separándolas por ; (punto y coma). De esta forma se ejecutan en secuencia, según el orden de izquierda a derecha.

### Obtener información

Los sistemas Unix disponen de un sistema de ayuda en línea. Basta invocar la orden man con un nombre de orden como argumento y el sistema mostrará la ayuda que tiene disponible para esa orden.

Ejemplo:

```
$  man ls
```

### Notación sobre órdenes

Para efectuar la descripción de las órdenes vamos a utilizar las siguientes convenciones:

[] Unos corchetes indican que el contenido de los corchetes es opcional.

a | b Una barra vertical nos indica que tenemos que escoger entre uno u otro, a ó b en este ejemplo.

plantilla Indica un nombre de fichero o directorio formado por caracteres normales y posiblemente los comodines * y ? vistos con anterioridad.

## Órdenes para ficheros y directorios

A continuación vamos a describir una serie de órdenes que se utilizan en la gestión y manipulación de ficheros y directorios por parte del usuario.

En este texto sólo se describen las opciones más frecuentes de cada orden. Es conveniente acostumbrarse a leer las páginas del manual de cada orden; en muchos casos nos podremos sorprender de las funcionalidades adicionales que admite.

## cat Mostrar contenido de un fichero

La orden cat muestra el contenido de un fichero por la salida estándar.

Uso:

```
cat [ruta]fichero
```

Si no ponemos ruta de acceso, no muestra los ficheros del directorio actual.

Ejemplos:

```
$ cat /etc/inittab
$ cat .profile
```

La orden cat se suele utilizar en muy diversos contextos. Su principal característica es que realiza un volcado de un fichero sin añadir ni quitar nada.

La orden cat nos puede servir para crear un fichero de texto sin tener que utilizar un editor de texto:

```
$ cat >borrar
hola
como
estas
(C-c)
$ cat borrar
hola
como
estas
$
```

Donde ponemos C-c significa pulsar la tecla Ctrl y sin soltarla pulsar la tecla c para tereminar de introducir líneas.

## ls Mostrar contenido de un directorio

Muestra el contenido de un directorio. Uso:

```
ls  [-abcdfgiklmnpqrstux] [--color][directorio...]
```

*ls* tiene más opciones, pero sólo vamos a citar las más importantes. Para obtener una información mas detallada consultar

```
$  man ls
```

Opciones:

- -a Se muestran todos los ficheros de los directorios, incluyendo los "invisibles"; es decir, aquéllos cuyos nombres empiezan por punto (‘.’).
- -l Se muestran el tipo, los permisos, el número de enlaces duros, el nombre del propietario, el del grupo, el tamaño en bytes, y una marca de tiempo. Ejemplos:

```
$ ls /etc
$ ls -la /etc
$ ls /dev
$ ls -la /dev
```

Nota: En Linux la opción --color hace que cada elemento del directorio aparezca de un color distinto según sea su tipo de fichero o directorio.

Comentarios:

Esta es una de las órdenes básicas de un sistema Unix, una de las más utilizadas. Si se quieren obtener ordenacines de los ficheros en su salida, en la página del manual explica tods las opciones. Si queremos ordenaciones más complejas, podemos combinarla con sort, aunque también trae sus propias opciones de ordenación.

## less Muestra un fichero de texto

La orden less muestra un fichero de texto línea a línea, a la vez que permite buscar palabras dentro de él.

Uso:

```
less [ruta]fichero
```

Ejemplo:

```
less /etc/hosts
```

Comentarios:

Esta orden no está presente en todos los Unix. Se utiliza con bastante frecuencia para ver el contenido de ficheros de texto que tienen un número de línea mayor que las que caben en la pantalla. Para salir de la visualización de un fichero mediante less utilice ls tecla 'q'.Véanse también las órdenes moremore, tailtail y headhead.

En muchas ocasiones es necesario hojear de forma cómoda algún fichero de configuración o de registro de incidencias y less nos permite hacerlo de una forma bastante cómoda.

## mkdir Crear un directorio

La orden mkdir crea un directorio

Uso:

```
mkdir [ruta]directorio
```

Si no ponemos ruta de acceso, *mkdir*, crea el directorio dentro de directorio actual. Si añadimos la opción -p creará todos los directorios necesarios hasta llegar al último en caso de que no existieran.

## rm Borrar un fichero o directorio

Elimina ficheros o directorios

Uso:

```
rm [-ir] [ruta]plantilla
```

- -i Pregunta si debe borrarse cada fichero. Si la respuesta no comienza por 'y' o por 'Y' (o quizá el equivalente local, en español 's' o 'S') no se borra.
- -r Borra recursivamente los contenidos de directorios. Borra un directorio y todo su contenido.

Comentarios:

Esta orden es muy peligrosa usada como *root* y sobre todo con la opción '-r'; un fichero borrado no se puede recuperar. Antes de usar *rm* es conveniente simular la orden de borrado con otra orden no destructiva como 'ls' para no llevarnos sorpresas desagradables. Por ejemplo si queremos borrar todos los ficheros que terminan en '.bq' podemos hacer:

```
lsls *.bq
```

y una vez que comprobamos que los ficheros que se van a borrar son los que realmente nos interesan, podremos

```
rm *.bq
```

Es necesario usar esta orden para mantener el sistema aseado, limpio de ficheros viejos que ocupan espacio en el sistema y nos llevan a confusiones, pero es una orden para usarla con precaución, sobre todo si estamos trabajando como root.

## cd Cambia el directorio actual

La orden cd cambia el directorio actual.

Uso:

```
cd [ruta]directorio
```

Si no ponemos argumentos, *cd*, cambia al directorio personal del usuario.

## mv Mueve o renombra

La orden mv mueve un fichero o directorio de un directorio a otro o le cambia el nombre.

Uso:

```
mv [ruta]origen [ruta]destino
```

Si en origen ponemos una plantilla como argumento, *mv*, necesita que destino sea un directorio.

## cp Copia ficheros y directorios

La orden *cp* copia un fichero o directorio en otro fichero y/o directorio distintos.

Uso:

```
cp [opciones] [ruta]origen [ruta]destino
```

- -i Pregunta si debe sobreescribir cada fichero destino que exista. Si la respuesta no comienza por 'y' o por 'Y' (o quizá el equivalente local, en español 's' o 'S') no se borra.
- -r Copia recursivamente los contenidos de directorios.
- -a Preserva los atributos del fichero copiado en la medida de lo posible.

*cp* tiene bastantes más opciones, por lo que se debería consultar la página correspondiente del manual.

## lpr Imprime un fichero

La orden lpr envía un fichero a la impresora.

Uso:

```
lpr [opciones][ruta]fichero
```

Cuando veamos la parte de administración se ampliará esta orden.

## pwd Imprimir el directorio actual

La orden pwd imprime el directorio actual.

Uso:

```
pwd
```

## ln Enlaza ficheros o directorios

La orden ln enlaza un fichero o directorio con otro fichero o directorio.

Uso:

```
ln [-sf] [ruta]origen [ruta]enlace
```

- -i Pregunta si debe sobreescribir cada fichero destino que exista. Si la respuesta no comienza por 'y' o por 'Y' (o quizá el equivalente local, en español 's' o 'S') no se borra.
- -s Crea un enlace simbólico.
- -f Elimina el fichero enlace si existía con anterioridad.

Por defecto, la orden *ln* crea un enlace duro, que consiste en crear una nueva entrada en un directorio que apunta a un i-nodo. Los enlaces duros no se permiten entre ficheros de sistemas de ficheros distintos, ya que todos los enlaces duros apuntan al mismo i-nodo, cosa imposible en dos sistemas de ficheros distintos (dos particiones distintas).

Si ponemos las opción -s entonces lo que creamos es un enlace simbólico, que consiste en un nuevo fichero que apunta al original, sea fichero o directorio. A diferencia del enlace duro, podemos enlazar ficheros entre diferentes sistemas de ficheros. El enlace simbólico crea un nuevo i-nodo mientras que el enlace duro no.

## Ejercicios con solución

Mostrar el contenido de todos los ficheros del directorio personal.

- $ cd cambiamos al directorio personal del usuario. Cada usuario tiene su directorio personal que suele ser */home/usuario*, donde usuario se corresponde al nombre que usamos para conectarnos.
  - $ pwd Nos aseguramos que estamos en él.
  - $ ls -la Mostramos la lista de ficheros del directorio.
  - $ cat fichero Mostramos el contenido de cada uno de los ficheros.
  - Repetimos las líneas anteriores tantas veces como sea necesario. Si por alguna circunstancia mostramos un fichero binario y los caracteres de la pantalla son ilegibles al terminar, debemos teclear *reset*. Podemos usar las flechas de cursor par buscar líneas lineas anteriores.

Copiar el fichero *.profile* en otro llamado perso.

- *$ cp .profile personal*

Crear un directorio llamado *prueba* en nuestro directorio personal.

- *$ cd*
  - *$ pwd*
  - *$ mkdir prueba* (Creamos el directorio )
  - *$ ls -la* (Verificamos la creación )

Expresar las rutas absoluta y relativa del directorio prueba que acabamos de crear. (Relativa al directorio actual).

- */home/usuario/prueba* Absoluta
  - *./prueba* Relativa

Copiar el fichero */home/usuario/perso* en el directorio *prueba*.

- Usando rutas absolutas *$ cp /home/usuario/perso /home/usuario/prueba*
  - Usando rutas relativas *$ cp /home/usuario/perso ./prueba* o *$ cp ./perso /home/usuario/prueba* o *$ cp ./perso prueba*

Cambiar al directorio */home/usuario/prueba*

- Usando rutas absolutas *$ cd /home/usuario/prueba*
  - Si estamos en */home/usuario $ cd prueba*
  - Si no estamos en */home/usuario $ cd $ cd ./prueba*

Copiar el fichero perso del directorio */home/usuario/prueba* con el nombre perso.nuevo (Suponemos que estamos en el directorio prueba, sin ver el ejercicio anterior).

- *$ cp perso perso.nuevo*
  - *$ ls -la*

Estando en el directorio prueba, copiar el fichero .profile en él con el nombre prof.nuevo

- *$ cp ../.profile prof.nuevo*
  - *$ ls -la*

Crear un enlace simbólico llamado *pro1*, al fichero *prof.nuevo*.

- *$ ln -s prof.nuevo pro1*

Verificar el enlace simbólico pro1 que acabamos de crear

- *$ ls -la* Comprobamos el tipo de fichero
  - *$ cat pro1* Comprobamos el contenido del fichero
  - Observamos que aparece un nuevo fichero con un tamaño muy pequeño. Sin embargo al mostrar el contenido aparecen los datos que corresponden al fichero al que apunta.

Crear un enlace duro llamado *prof.d1*, al fichero *prof.nuevo*.

- *$ ln prof.nuevo prof.d1*

Crear otro enlace enlace duro llamado *prof.d2*, al fichero *prof.d1*.

- *$ ln prof.nuevo prof.d2*
  - *$ ls -lai*
  - Con esta operación estamos creando una nueva entrada al directorio que corresponde a unos datos existentes en disco previamente. Al añadir la opción -i a la orden ls también nos informa del número de *i-nodo* y podemos observar que son el mismo.

Borrar el fichero *prof.d1* y verificar los demás

- *$ rm prof.d1*
  - *$ ls -lai*
  - Podemos observar como tras borrar *prof.d1* seguimos teniendo el fichero *prof.d2* y con el mismo *i-nodo*.

Crear un directorio dentro de prueba llamado *src*.

- *$ mkdir src*
  - *$ ls -la* Verificamos la creación

Crear un enlace simbólico llamado *fuente*, al directorio *src*.

- *$ ln -s src fuente*

Copiar el fichero *perso.nuevo*, al directorio *src*.

- $ cp perso.nuevo src

Copiar el fichero pro1, al directorio *fuentes*.

- *$ cp pro1 fuentes*

Verificar los contenidos de los directorios *src* y *fuentes*.

- *$ ls -la fuentes*
  - *$ ls -la src*

Cambiar al directorio superior.

- *$ cd ..*

Cambiar directamente al directorio fuentes.

- *$ cd pruebas/fuentes*

Borrar el directorio src.

- *$ cd ..*
  - *$ rm -r src*

Verificar el estado del enlace simbólico *fuentes*. Borrar el enlace.

- *$ ls -la fuentes*
  - Observamos como la orden ls -la tiene ahora un significado distinto a cuando existía el directorio src. Observa el error que se produce cuando hacemos cat fuentes.
  - *$ rm fuentes para borrarlo*

Copiar el fichero */home/usuario/perso* en el fichero *nuevo*.

- *$ cd*

- *$ cp perso nuevo*

Cambiar el nombre del fichero nuevo a *viejo*.

- *$ mv nuevo viejo*
- *$ ls -la*

Crear un directorio llamado *practica* en el directorio personal del usuario.

- *$ cd*
- *$ mkdir practica*

Mover el fichero *viejo* al directorio *practica*

- *$ mv viejo practica*

Mostrar la lista de usuarios del sistema

- *$ cat /etc/passwd*

Mostrar la lista de grupos del sistema

- *$ cat /etc/group*

Copiar el fichero */etc/hosts* en el directorio personal

- *$ cp /etc/hosts .*

Mostrar el fichero *hosts* del directorio personal

- *$ cat hosts .*

Mostrar el contenido del directorio */tmp*. Observar permisos y propietarios.

- *$ ls -la /tmp*

Copiar el fichero /etc/inittab en el directorio personal

- *$ cp /etc/inittab .*

Cambiar de nombre al fichero *inittab* del directorio personal y llamarlo *tabla.inicio*

- *$ mv inittab tabla.inicio*

Crear un directorio llamado *programas* y dentro de él, otros cuatro directorios llamados *src*, *lib*, *bin* y *include*

- *$ mkdir programas*
  - *$ mkdir programas/src*
  - *$ mkdir programas/lib*
  - *$ mkdir programas/bin*
  - *$ mkdir programas/include*

Cambiar al directorio *src*

- *$ cd programas/src*

Cambiar al directorio superior

- *$ cd ..*

Crear un directorio llamado config en el directorio personal

- *$ cd*
  - *$ mkdir config*

Copiar en directorio config todos los ficheros de */etc* que empiecen por s.

- *$ cp /etc/s* ./config*

Mostrar el contenido del directorio *config*

- *$ ls -la ./config*

Mostrar el contenido del directorio *config*

- *$ ls -la ./config*

Borrar el contenido del directorio config

- *$ rm ./config/**

Borrar el directorio *config*

- *$ rm -r ./config*

Crear un enlace simbólico con el directorio */tmp* llamado *temp*.

- *$ ln -s /tmp temp*

Ejecutar cd *~/programas* y verificar el directorio actual.

- *$ cd ~/programas*
  - *$ pwd*

Ejecutar *cd ~/programas/bin* y verificar el directorio actual.

- *$ cd ~/programas/bin*
  - *$ pwd*
  - Podemos observar como el símbolo ~ hace referencia al directorio personal del usuario.

# Comprimir, descomprimir y agrupar ficheros

## Comprimir ficheros y uso de ficheros compromidos

La forma habitual de transmitir y almacenar información es hacerlo con ficheros comprimidos. Se ahorra ancho de banda en las transmisiones y en el almacenamiento. Los mecanismos de compresión son distintos, y en muchos casos incompatibles entre sí, pero Linux dispone de utilidades para gestionar todos ellos.

### gzip

La orden gzip comprime un fichero y es un estándar en sistemas Linux. Su uso es muy simple:

```
$ gzip fichero
```

y se crea un fichero llamado *fichero.gz* comprimido.

Esta orden normalmente se instala en todos los sistemas Linux

gunzip

Se utiliza para descomprimir un fichero comprimido con gzip. Su uso es el lógico:

```
$ gunzip fichero.gz
```

zcat

La orden *zcat* es idéntica a *cat*, salvo que trabaja directamente con ficheros comprimidos con *gzip*.

zless

Esta orden es idéntica a *less*salvo que trabaja directamente con ficheros comprimidos con *gzip*.

### bzip2

Esta orden comprime ficheros. Su uso es idéntico a *gzip*, pero es algo más efectiva que *gzip*, genera ficheros comprimidos algo más pequeños. Esta orden genera fichros con extensión *.bz2*.

Este compresor puede que no se instale con Linux si no se indica explícitamente.

### bunzip2

Esta es la orden que nos permite descomprimir ficheros comprimidos con *bzip2*. Los ficheros comprimidos com *bzip2* tienen la extensión *.bz2*.

### zip

Esta es la orden se utiliza para comprimir ficheros en formato *zip*. El uso es ligaramente distinto a las órdenes anteriores:

```
$ zip ficheros-comprimido lista_ficheros
```

Es decir, primero ponemos el nombre del fichero comprimido y luego la lista de ficheros que queremos comprimir.

Los ficheros comprimidos con zip tienen extensión *.zip*.

Este compresor puede que no se instale con Linux si no se indica explícitamente.

### unzip

Esta orden descomprime ficheros previamente comprimidos con *zip*.

## Agrupar y desagrupar ficheros: tar

En las órdenes anteriores hemos visto como comprimir ficheros individuales, pero en muchas ocasiones nos interesa agrupar en un archivo diferentes ficheros, por ejemplo todos los contenidos en un directorio.

También es habitual en linux distribuir los programas con todos sus ficheros agrupados y comprimido, por lo que será necesario saber como descomprimir y desagrupar los ficheros.

Estas operaciones las realiza la orden tar (tape archive); las extensiones que utiliza esta orden para indicar un archivo de ficharos es .tar.

### Agrupar ficheros

Para agrupar una serie de ficheros en un archivo pondremos:

```
$ tar cf archivo.tar lista de ficheros
```

Si en lugar de poner "lista de ficheros" ponemos un directorio agrupará todos los ficheros contenidos en el directorio y conservando la estructura del subárbol de directorios.

La orden ejecutada de esta forma simplemente comprime los ficheros, pero sin realizar ningún tipo de compresión.

### Agrupar ficheros y comprimir

Para agrupar una serie de ficheros en un archivo y comprimirlo con *gzip* simultáneamente pondremos:

```
$ tar cfz archivo.tar.gz lista de ficheros
```

Para agrupar una serie de ficheros en un archivo y comprimirlo con *bzip2* simultáneamente pondremos:

```
$ tar cfI archivo.tar.bz2 lista de ficheros.
```

Si en lugar de poner "lista de ficheros" ponemos un directorio agrupará todos los ficheros contenidos en el directorio y conservando la estructura del subárbol de directorios.

### Desagrupar ficheros

Para desagrupar una serie de ficheros en un archivo pondremos:

```
$ tar xvf archivo.tar
```

### Desagrupra ficheros y descomprimir

Para desagrupar una serie de ficheros en un archivo comprimido con *gzip* pondremos:

```
$ tar xvfz archivo.tar.gz
```

Si el archivo estuviera comprimido con *bzip2* pondríamos:

```
$ tar xvfI archivo.tar.bz2
```

En ambos casos podríamos descomprimir el fichero con *gunzip* o con *gunzip,* según el caso y luego desagrupar el archivo no comprimido como se indicaba más arriba.

## Otras órdenes de usuario

### id Información sobre el usuario

La orden id imprime los identificadores de grupo y usuario reales del usuario. Esta orden dispone de diversos argumentos que se pueden consultar en la correspondiente página del manual.

```
Uso:    id
```

Ejemplo:

```
$ id  uid=500(fabrega) gid=500(fabrega) groups=500(fabrega),4(adm),100(users),505(usuwi
```

### passwd Modifica la clave

Permite a un usuario modificar su clave simplemente ejecutándola. El root puede ejecutar esta orden añadiendo como argumento el nombre de un usuario para cambiar su clave. Los cambios de clave implican modificaciones del fichero */etc/passwd* (o */etc/shadow* en su caso). Uso:

```
   passwd [usuario]
```

## man  Proporciona información

La orden *man* proporciona información sobre sobre una orden, programa o función del lenguaje C. La información se guarda en el directorio */usr/man*. Hay una serie de variables de entorno que permiten modificar el comportamiento de la orden man: La variable *LANG* define el idioma en que suministra la información. En nuestro caso nos interesará poner *LANG=es*. Si una página no existe en el idioma, se suministra en inglés. Para que aparezcan acentos y eñes tendremos que usar la variable *LESSCHARSET* asignándole el valor *latin1*.

Uso:

```
man [sección] orden
```

Ejemplos:

```
      $ man ls
$ man 2 mount
$ man mount
```

## who  Información sobre usuarios conectados

La orden who proporciona información sobre los usuarios conectados al sistema en un momento dado. Esta orden dispone de diversos argumentos que se pueden consultar en la correspondiente página del manual. Uso:

```
who
```

Por ejemplo, tendremos:

```
$ who
fabrega  tty1     Ene 19 14:42 root     tty2     Ene 30 20:17
```

## whoami  Información sobre el usuario

Esta orden es equivalente a *id -un*.

Esta orden es útil cuando estamos trabajando con varios usuarios simultáneamente y queremos saber en un momento dado cual es el que estamos utiliando.

```
$ whoami
pfabrega
```

## write  Envía un mensaje a un usuario

Uso:

```
write usuario
```

La orden *write* envía un mensaje a un usuario conectado a la misma máquina en otro terminal distinto.

## mesg  Activa/Desactiva la recepción de mensajes

Uso:

```
mesg y|n
```

Con esta orden podemos impedir que otro usuario escriba en el terminal que estamos utilizando. En particular desactiva los mensajes enviados con *write*. Observe como se modifican los permisos del terminal correspondiente en */dev/*.

## mail  Envía un mensaje de correo electrónico

Uso:

```
mail usuario[@maquina]
```

Además esta orden también se utiliza para leer el correo que se recibe, en este caso llamándola como mail.

Véase también *mailx*.

Para obtener mas detalles consulte la correspondiente página del manual.

## date  Muestra las hora y fecha actuales

Uso:

```
date [opciones]
```

Ejemplo:

```
$ date
mié ene  6 19:53:26 CEST 1999
```

Las opciones es una cadena de formato constituida por caracteres fijos que aparecen tal cual se ponen y otros caracteres precedidos por el carácter % que se sustituyen por el valor correspondiente de la fecha La cadena de formato va precedida de un signo +. Algunos valores admitidos son:

%y año con dos dígitos

%Y año con cuatro dígitos

%m número del mes

%d número del día del mes

por ejemplo:

```
$ date "+%Y%m%d"
20011207
$ date "+ año %y mes %m dia %d"
```

```
año 01 mes 12 dia 07
```

Para obtener mas detalles consulte la correspondiente página del manual.

Comentario: Esta orden es interesante para generar nombres de ficheros que contengan la fecha actual, para guardar copias de seguridad, por ejemplo.

## echo  Muestra en pantalla el resto de la línea

Uso:

```
echo  [-ne] [cadena ...]
```

-n Elimina los retornos de carro finales

-e Usa el carácter de barra invertida (\ ) como carácter de escape. Es decir "\ n" se considera como un salto de línea. La orden echo interpreta cualquier cadena que comience por «$» como una variable. Las variables se expanden cuando la cadena va encerrada entre comillas dobles (""). Todo lo que vaya comprendido entre comillas simples se interpreta tal cual. También podemos imprimir un carácter $ protegiéndolo con el carácter de escape, que vimos anteriormente (\ ). (Véanse los metacaracteres de la shell).

## sort  Ordena el contenido de un fichero

Uso:

```
sort fichero
```

Es conveniente mirar las opciones de las que dispone la orden sort si queremos una ordenación con alguna característica concreta.

## more  Muestra un fichero

Uso:

```
more fichero
```

Esta orden es muy parecida a *less*

## cal  Muestra un calendario

Uso:

```
cal [mes] año
```

Por ejemplo, tendremos la siguiente salida

```
$ cal 1 99
```

```
     enero de 99
do lu ma mi ju vi sá
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

## expr  Evalúa una expresión entera

Uso:

```
expr expresión
```

La orden *expr* evalúa una expresión entera y la muestra. Si la expresión es lógica devuelve un 1 en caso de ser verdadera y un 0 en caso de ser falsa.

Los operadores que admite la orden *expr* son los siguientes:

Aritméticos *, +, -, /, %

Comparar cadenas :

Lógicos <, <=, =, !=, >=, >

And y Or &, |

Hay que tener en cuenta que ciertos caracteres tienen un significado especial para la shell; cuando los usemos tendremos que protegerlos con el carácter de escape \ .

Ejemplos:

```
$ expr 20 + 15
$ expr 30 * 6
$ expr 29 % 3
$ expr 101 / 7
$ expr 5  \  < 8
$ expr 5  \  > 8
```

## diff  Muestra diferencias entre ficheros

Uso:

```
diff [opciones] fichero-origen fichero-destino
```

Los argumentos también pueden ser dos directorios, en cuyo caso compara sus contenidos. Consultar la página del manual para tener más información.

## find  Localiza ficheros

Uso:

```
find [ruta] [expresión]
```

La orden find se utiliza para localizar ficheros contenidos en los distintos discos del sistema. Esta orden dispone de muchas opciones por lo que es conveniente consultar la página del manual. Entre las opciones disponibles de find podemos citar:

*-name expresión* indica el nombre de fichero que queremos buscar. Puede ser un nombre fijo o una plantilla formada por * y ?, en este último caso, la plantilla deberá estar comprendida entre comillas dobles para evitar que la shell las interprete antes de llamar a la orden find.

*-type tipo* donde tipo puede ser b, c d o f para referirnos a dispositivos de bloque (b), carácter (c), directorio (d) o fichero regular (f).

*-exec orden* ejecuta una orden para cada fichero encontrado. (Véanse los ejemplos para los detalles de la sintaxis).

*-o* realiza un O lógico entre dos opciones.

*-user usuario* busca ficheros propiedad del usuario.

*-group grupo* igual que user pero para el grupo.

```
$ find . -name .profile
$ find / -name core -type f -exec rm {} \;
$ find /tmp  \(-user root -o -user pepe \)
$ find / -name "print*"
```

La orden *find* es muy útil para localizar ficheros con características muy concretas, por ejemplo todo tipo de permisos (opción *-perm*) modificados en los últimos días (*-mtime*), accedidos en los últimos días (*-atime*) etc. Es muy útil sobre todo en aspectos de seguridad del sistema.

## ps: Muestra lista de procesos

Uso:

```
ps [axu...]
```

La orden *ps* muestra la lista de procesos del sistema y algunas de sus características, dependiendo de las opciones que le añadamos. Entre las múltiples opciones disponibles vamos a citar:

- -a Muestra también los procesos de otros usuarios
- -x muestra procesos que no están controlados por ninguna terminal
- -u formato usuario: muestra el usuario y la hora de inicio Pero no sólo estas opciones están disponibles, también hay otras que pueden ser muy útiles en ciertas circunstancias. Es conveniente consultar la página del manual.

Este sería un ejemplo, sin opciones:

```
$ ps
PID TTY          TIME CMD
26861 ttyp4    00:00:00 bash
4767 ttyp4    00:00:00 ps
```

Otro ejemplo podría ser

```
$ ps x    PID TTY      STAT   TIME COMMAND
686 tty1     S      0:00 -bash
26812 tty1     S      0:00 sh /usr/X11R6/bin/stkde -- -bpp 32 -cc 4
26813 tty1     S      0:00 xinit /etc/X11/xinit/xinitrc.kde -- -bpp 32 -cc 4
26817 tty1     S      0:15 kwm 26836 tty1     S      0:00 kaudioserver
26837 tty1     S      0:00 kwmsound 26838 tty1     S      0:26 kfm
26839 tty1     S      0:00 krootwm 26840 tty1     S      0:26 kpanel
```

## sleep: Genera un proceso durante cierto tiempo

Uso:

```
sleep numero[smhd]
```

La orden *sleep* genera un proceso que dura el tiempo especificado. El tiempo se puede especificar en segundos (s), que es el valor que se toma si se omite la letra. También en minutos (m), horas (h) o días (d).

Esta orden puede parecer una tontería, pero se puede combinar con otras órdenes para demorar cierto tiempo el comienzo de su ejecución. Ejemplo:

```
# sleep 2h; ppp-off
```

## stty: Parámetros del terminal

Uso:

```
stty [valores...]
```

*stty* toma una serie de argumentos que modifican las características del terminal. Si añadimos un '[-]' antes del argumento, la característica se deshabilita, si no se habilita.

Por ejemplo:

```
$ stty sane
```

repondría en el terminal sus valores originales.

## head: Muestra las primeras línea de un fichero

Uso:

```
head [opciones...] fichero
```

La orden head nos permite ver las primeras líneas de un fichero, y nos puede ser útil para identificar ficheros por su contenido.

Por ejemplo, queremos saber la versión del fichero */etc/sendmail.cf* y sabemos que está en las tres primeras líneas. Entonces pondríamos:

```
$ head -3 /etc/sendmail.cf
```

## tail: Muestra las últimas línea de un fichero

Uso:

```
tail [opciones...] fichero
```

Por ejemplo, para mostrar las dos últimas línea del fichero /etc/group pondríamos:

```
$ tail -s /etc/group
```

La orden tail nos permite ver las primeras líneas de un fichero, y nos puede ser útil para ir viendo las incidencias que se anotan en un fichero de registro. Por ejemplo, es frecuente en ciertas situaciones que el administrador tenga una consola virtual para ejecutar la orden:

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

Con la opción -f hacemos que tail muestre las líneas según vayan escribiendo en el fichero */var/log/messages*. Para cancelar la ejecución de esta orden (con la opción -f) tendremos que pulsar Ctrl C.

Comentario:

Esta orden se utiliza con bastante frecuencia para ir mostrando en pantalla ficheros de registros de incidencias par ir viendo en el momento lo que ocurre con el sistema.

## touch: Actualiza las fechas de un fichero

Uso:

```
touch [opciones...] fichero
```

La orden touch modifica las fechas de acceso y modificación de un fichero. En el caso de que el fichero no exista, la orden touch crea un fichero vacío.

touch dispone de diversas opciones. Para verlas, consultar la página del manual.

Muchos programas que sólo pueden tener una copia ejecutándose en el sistema utilizan touch para crear un fichero que indica que el proceso está activo; antes de terminar la ejecución se debe borrar este fichero.

## tty: Muestra el terminal

Uso:

```
tty
```

La orden tty muestra la terminal actualmente conectada a la salida estándar.

## wc: Cuenta información sobre ficheros

Uso:

```
wc [opciones...] fichero
```

La orden wc cuenta, sobre un fichero de texto, líneas, caracteres y palabras. No nos confundamos, *wc* significa *word count*

Si no ponemos opciones, *wc* contará líneas, palabras y caracteres. Si queremos limitar las operaciones de *wc* disponemos de las siguientes opciones:

- -c Cuenta caracteres
- -w Cuenta palabras
- -l Cuenta líneas

## Ejemplos

Que un usuario modifique su contraseña de acceso

```
$ passwd
```

La orden *passwd* modifica la contraseña del usuario que la ejecuta. El administrador puede usarla poniendo a continuación un nombre de usuario para modificar las contraseñas de otras cuentas del sistema.

Averiguar los usuarios que están en este momento conectados al sistema.

```
$ who
```

Averiguar qué usuario tengo activo en una determinada consola o terminal.

Para resolver esta cuestión nos podemos basar en el caso anterior. La orden who también muestra la consola junto al nombre de usuario, por lo que bastaría poner:

```
$ who
```

Enviar un mensaje al usuario juan que está actualmente conectado.

```
$ write juan "Dame la dirección de corre de Antonio, please"
```

La orden *write* sólo es válida cuando el otro usuario está conectado al sistema. Si quisiéramos enviar mensajes a usuarios que no están conectados tendríamos que usar el correo electrónico.

Desactivar la recepción de mensaje en nuestra terminal

```
$ mesg n
```

La orden *mesg* básicamente lo que hace es modificar los permisos del correspondiente dispositivo */dev/ttyN*

Mostrar las fecha y hora actuales

```
$ date
```

Mostrar ordenados los ficheros */etc/passwd* y */etc/group*

```
$ sort /etc/passwd      $ sort /etc/group
```

Mostrar el fichero */etc/services* pantalla a pantalla

```
$ more /etc/services
```

Mostrar el calendario del 1 de enero de 1999 */etc/group*

```
$ cal 1 1999
```

Calcular las siguientes operaciones:

110+38 resto de 100/7 si 3<8 si 8<3 si 8>3

Puestas en el mismo orden quedarían:

```
   expr 110 + 38
 expr 100 % 7
  expr 3  \  < 8
  expr 8  \  < 3
  expr 8  \  > 3
```

Aquí podemos observar como hemos incluido el carácter \ para proteger los símbolos < y >. Estos dos símbolo tienen un significado propio, por lo que es necesario "protegerlos" para evitar que tengan este comportamiento (redirigir las entrada y salida estándares). También hay que tener en cuenta los espacios.

Localizar los ficheros *resolv.conf*, *profile*, *hosts.allow* y *sendmail.cf*.

```
   $ find / -name resolv.conf -type f
$ find / -name profile -type f
$ find / -name hosts.allow
$ find / -name sendmail.cf -type f
```

Localizar todos los directorios llamados usr y bin.

```
    $ find / -name usr -type d
$ find / -name bin -type d
```

Localizar el fichero llamado aliases y mostrarlo en pantalla.

```
$ find /etc -name aliases -type f -exec cat {} \ ;
```

Mostrar la lista de procesos propiedad del usuario que ejecute la orden.

```
$ ps
```

Mostrar la lista de todos los procesos activos en la máquina, indicando el usuario que es su propietario.

```
$ ps axu
```

Poner a dormir un proceso durante 5 segundos.

```
$ sleep 5
```

Mostrar las cinco primeras líneas del fichero */etc/inetd.conf*.

```
$ head -n 5 /etc/inetd.conf
```

Mostrar las últimas líneas del fichero /etc/inittab.

```
$ tail /etc/inittab
```

Crear un fichero vacío llamado nuevo en nuestro directorio personal.

```
$ cd
$ touch nuevo
```

Contar las líneas, palabras y caracteres que contiene el fichero /etc/passwd.

```
$ wc -l /etc/passwd
$ wc -w /etc/passwd
$ wc -c /etc/passwd
```

# Gestión de procesos

## introducción

Como ya debemos saber, un proceso es un programa en ejecución con recursos asignados. También sabemos que un proceso puede tener distintos estados que se pueden clasificar según distintos criterios, algunos de los cuales no son incompatibles entre sí. Como ahora lo que nos interesa es el aspecto puramente práctico, vamos a distinguir tres estados:

- Primer plano: Un proceso que se ejecuta bloqueando para él la terminal desde la que e lanzó. Un proceso se lanza en primer planos simplemente introduciendo su nombre (y la ruta de acceso si fuera necesario) en el indicador de la línea de órdenes y pulsando intro.
- Segundo plano: Un proceso que se ejecuta sin bloquear la terminal, aunque sí puede escribir en ella los resultados de su ejecución. Un proceso se lanza en segundo plano poniendo al final de la línea de órdenes el símbolo & separado por al menos un espacio del nombre del programa.
- Detenido: Podemos detener un proceso y que se quede en espera en el sistema hasta que demos la orden para que continúe su ejecución. En el presente texto nos vamos a referir a procesos detenidos por orden directa del usuario. No vamos a hacer referencia a procesos suspendidos por causas internas del sistema operativo. En general disponemos de la orden ps que nos proporciona información sobre los procesos, como por ejemplo hora de inicio, uso de memoria, estado de ejecución, propietario y otros detalles.

Además tenemos que tener en cuenta otra característica: cada proceso tiene un propietario que generalmente es el usuario que lo ejecuta. Además al proceso se le aplican los mismos permisos que tenga el usuario propietario, es decir el proceso sólo podrá acceder a la información a la que pueda acceder el propio usuario con sus permisos.

## Operaciones con procesos

Operaciones con procesos en primer plano

Con un proceso en primer plano podemos realizar dos acciones desde la terminal que tiene asociada.

- Matar el proceso: C-c (Ctrl-c ) Sólo podremos matar procesos sobre los que tengamos permiso. Si intentamos matar el proceso de otro usuario el sistema no nos lo permitirá, salvo a root.
- Parar el proceso: C-z (Ctrl-z) En el primer caso se cancela el proceso y se liberan todos los recursos que tuviera asignados. En el segundo caso sólo se detiene la ejecución del proceso, conservando su estado y sus recursos para poder continuar en el momento que se dé la orden adecuada.

Ejemplo: Vamos a lanzar un proceso que dura 20 segundos y a continuación lo vamos a matar:

*$ ps* En primer lugar vemos la lista de procesos.

*$ sleep 20* Creamos el proceso y observamos como la terminal se queda bloqueada durante la ejecución.

*Ctlr-c* Matamos el proceso y observamos como se devuelve el control de la terminal y aparece el indicador de la línea de órdenes.

*$ ps* Y ahora observamos como los procesos que aparecen son los mismos que al principio.

Ahora vamos a repetir la operación, pero deteniendo el proceso

*$ ps* En primer lugar vemos la lista de procesos.

*$ sleep 20* Creamos el proceso y observamos como la terminal se queda bloqueada durante la ejecución.

*C-z (Control-z)* Detenemos el proceso y observamos como se devuelve el control de la terminal y aparece el indicador de la línea de órdenes.

*$ ps* Y ahora observamos como entre los procesos aparece ahora *sleep 20*.

Envío de señales a procesos

Las órdenes kill y killall se utilizan para enviar señales a un proceso. Las señales más habituales son:

SIGHUP (1) Colgar. Ciertos procesos utilizan la señal SIGHUP para indicar al proceso que relea su fichero de configuración. Este es el caso, por ejemplo, de los procesos init y inetd.

SIGCONT (18) Continuar. Cuando un proceso detenido recibe esta señal continúa su ejecución.

SIGSTOP (19) Cuando un proceso en estado preparado recibe esta señal se detiene.

SIGINT (2) Termina un proceso y desaparece.

SIGTERM (15) Mata un proceso de forma controlada.

SIGKILL (9) Esta señal mata un proceso incondicionalmente. Para enviar una señal a un proceso disponemos de dos órdenes que realizan esa labor:

Orden kill Esta orden la podemos usar de la siguiente forma:

```
$ kill -señal lista_pid_proceso
```

donde el valor de lista_pid_proceso tendremos que averiguarlos utilizando la orden ps. En cuanto a señal puede ser bien su valor numérico o bien su valor simbólico, omitiendo el SIG inicial. Por ejemplo:

```
$ kill -9 337
```

o lo que sería lo mismo

```
$ kill -KILL 337
```

Orden killall Esta orden es ligeramente diferente a la orden *kill* por dos motivos; en primer lugar utiliza el nombre de proceso en lugar del *pid*, y además le envía la señal a todos los procesos que tengan el mismo nombre. Es decir:

```
$ killall -señal nombre_proceso
```

Por lo demás, su comportamiento es idéntico, por lo que serían equivalentes

```
$ kill -HUP 1
$ killall -HUP init
```

al haber un único proceso init, con pid igual a 1.

También hay que tener en cuenta que todos los procesos no están preparados para recibir todas las señales. Algunas señales siguen un tratamiento estándar, otras las gestiona el propio proceso y otras son simplemente ignoradas.

Ejemplos:

```
$ ps
$ sleep 100 (pulsar C-z)
$ ps
$ kill -CONT pid    # pid se sustituye por el valor obtenido en ps    (pulsar C-z)
$ kill -9 pid
$ ps     $ killall -HUP init
```

y ahora se puede repetir lo mismo con killall. Por ejemplo:

```
$ ps
$ sleep 100
$ C-z
$ ps
$ killall -CONT pid
$ C-z
$ killall -9 pid
$ ps
```

Procesos en segundo plano

Hasta ahora hemos visto una serie de pasos para poder enviar un proceso para que continúe su ejecución en segundo plano; le enviamos una señal para que se detenga y luego otra para que continúe su ejecución. Pero este proceso es incómodo, tenemos una forma más simple de enviar un programa para que se ejecute directamente en segundo plano. Esto es con el operador & tras el nombre del programa y su argumentos. Un ejemplo simple de esta acción sería:

```
$ sleep 200 &
$ ps
```

Al lanzar el proceso, nos aparece por pantalla algo parecido a:

```
[1]   2 035
```

donde: *[1]* es el número de trabajo

*2035* es el número de proceso.

Podemos observar como el indicador de la línea de órdenes no aparece inmediatamente. Sin embargo esto no supone que la terminal de salida no sea desde la que se lanzó el proceso; es decir, el proceso seguirá mostrando su salida en la pantalla desde la que dimos la orden de ejecución.

Ahora, a ese proceso en segundo plano podemos enviarle señales con las órdenes kill y killall como hicimos antes.

Ejemplos:

```
   $ sleep 200 &
$ sleep 201 &
$ ps
```

y podemos observar como los procesos continúan activos.

```
$ sleep 10 &
```

Ahora esperamos 10 segundos y el sistema nos avisa que ha terminado.

```
[3]+  Done                    sleep 10
```

## Gestión de trabajos

Cuando lanzamos un proceso en segundo plano obtenemos un PID y un número de trabajo. El PID es el número de proceso y es es método que utiliza el sistema operativo para identificar de forma única al proceso. En cambio el número de trabajo es un identificador de uno o varios procesos correspondientes a un usuario.

Anteriormente vimos que C-z detiene (suspende) un proceso y lo deja en segundo plano. También vimos como podíamos enviarle un señal para que continuara su ejecución en segundo plano. Ahora lo que vamos ver son los mecanismos para realizar una gestión más completa de esos trabajos.

### fg (foreground)

La orden *fg* se utiliza como:

```
$ fg [%num_trabajo]
```

y se utiliza para traer a primer plano un trabajo que está en segundo plano, bien esté activo o bien esté detenido.

Ejemplo:

```
   $ sleep 200  (y pulsar C-z)
$ sleep 300  (y pulsar C-z)
$ fg %1      (y pulsar C-z)
$ fg %2
```

### bg (background)

Hasta ahora hemos alternado entre un proceso detenido y un proceso en primer plano, y podíamos hacer que continuara enviándole una señal. Tenemos otra forma para hacer que un proceso detenido en segundo plano continúe su ejecución. La orden *bg* se utiliza como:

```
$ bg [%num_trabajo]
```

y se utiliza para poner en ejecución en segundo plano un trabajo que está en segundo plano detenido.

Ejemplo:

```
$ sleep 200  (y pulsar C-z)
```

```
$ ps
$ bg
$ sleep 201 &
$ ps     $ fg %2      (y pulsar C-z)
$ fg %1      (y pulsar C-z)
$ bg %1
$ bg %2
$ ps
```

### jobs

Con la orden *jobs* podemos obtener una lista de los trabajos que hemos lanzado en el sistema. La orden *jobs* se utiliza como:

```
$ jobs
```

Al usarla nos aparece algo como

```
$ jobs
[1] Running sleep 100 &
[2] Running sleep 101 &
[3]- Running sleep 102 &
[4]+ Running sleep 103 &
```

donde entre corchetes tenemos el número de trabajo, y los signos + y menos indican:

+ El trabajo es el primero de la lista

- El trabajo es el segundo de la lista

Ejemplo:

```
     $ sleep 500 &
    $ sleep 450 ( pulsamos Ctlr-z)
[2]+ Stopped sleep 450
    $ jobs
[1]- Running sleep 500 &
[2]+ Stopped sleep 450
    $ bg
[2]+ sleep 450 &
```

y vemos como obtenemos la lista de trabajos y su estado

### kill %

Con la orden

```
$ kill %nº_trabajo ...
```

podemos matar un trabajo en ejecución.

### nohup

La orden *nohup* lanza un proceso y lo independiza del terminal que estamos usando. Los procesos se organizan de forma jerárquica, de forma que si abandonamos la shell que nos conectó al sistema (abandonamos la sesión de trabajo) automáticamente se matarán todos los procesos que dependan de ella. Pero en muchas ocasiones no puede interesar lanzar un proceso y dejarlo en ejecución aun cuando hayamos cerrado la sesión de trabajo. Para esto se usa la orden *nohup*. Esta orden se usa como:

```
$ nohup orden [argumentos]
```

## Entrada y salida

### Dispositivos estándares

Cualquier proceso que se lanza, por omisión, dispone de tres ficheros abiertos que son sus canales de entrada y salida estándares y salida de errores. La entrada estándar, el canal 0, también llamado *stdin*, por omisión se asocia al teclado. La salida estándar, el canal 1 o *stdout* por omisión se asocia a la pantalla (el terminal). La salida de errores, el canal 2 llamado *stderr* se asocia por omisión también a la pantalla al igual que la salida estándar.

Resumiendo, existen tres ficheros estándares asociados al teclado y a la pantalla.

Cada proceso tiene la posibilidad de alterar estos valores estándares y usar como entrada o salida aquéllos que se defina. También nosotros, desde la línea de órdenes tenemos la posibilidad de alterar ese comportamiento como veremos más adelante.

### Otros dispositivos

En el sistema existen otros dispositivos asociados a distintos componentes de hardware del sistema. Los dispositivos se almacenan como ficheros en el directorio */dev*. En la parte d administración del sistema se verán con más detalle los dispositivos, ahora no limitamos a una breve visión a nivel de usuario.

Como ejemplo de dispositivos podemos citar:

#### Discos duros IDE

```
/dev/hdXY
     ^ ^
  | |Y número de partición
  | X Número IDE a=1 master b=1 esclavo c=2 master d=2 esclavo
     hd = hard disk
```

#### Discos flexibles

```
/dev/fdX
      ^ ^
   | |X número de unidad 0 = primera 1, = segunda
   | fd = floppy disk
```

### Puertos serie

```
/dev/ttySn
      ^ ^
    | |n número de puerto
    | S = serie
    tty = terminal
```

### Otros dispositivos

*/dev/psaux* puerto ps2

*/dev/audio* sonido

/dev/dsp sonido

*/dev/lp0* primer puerto de impresora

*/dev/null* dispositivo nulo

Este último dispositivo se utiliza para descartar salidas. Todo lo que se envía a */dev/null* simplemente se pierde.

## Redirección

En muchos casos nos va a interesar alterar las entradas y salidas estándares de un programa para que la información venga de otro origen distinto del habitual, o que la salida en vez de mostrarse en la pantalla se guarde en un dispositivo o fichero.

Para realizar esta redirección tendremos que usar los metacaracteres:

> redirige la salida la salida estándar a un fichero nuevo. Si el fichero existe se borra el contenido previo.

< redirige la entrada estándar por un fichero.

2> redirige la salida de errores a un fichero nuevo. Si el fichero existe se borra el contenido previo.

>> redirige la salida la salida estándar añadiéndola al final de un fichero. Si el fichero no existe lo crea.

2>> redirige la salida de errores añadiéndola al final de un fichero. Si el fichero no existe lo crea.

Ejemplos:

Realizar los siguientes ejemplo en la shell:

```
$ echo "Primera Línea" > nuevo
$ cat nuevo
$ echo "Segunda línea" >> nuevo
$ cat nuevo
$ cat nuevo > nuevo.1
$ cat nuevo.1
$ echo "tercera linea" > nuevo
$ cat nuevo
$ mail usuario -s "prueba con mail" <nuevo.1
$ aaa
$ aaa > nuevo.2
```

```
$ cat nuevo.2
$ aaa 2> nuevo.2
$ cat nuevo.2
$ cat <<! >nuevo.3
escribir
varias
líneas
y terminar con un ! sólo en una línea
!
$ cat nuevo.3
```

## Tuberías o pipes

Anteriormente vimos que cada proceso dispone de tres canales estándar para comunicarse con el exterior, las entrada y salida estándares y la salida de errores. En cualquier sistema Unix se puede hacer que la salida de una determinada orden sea la entrada estándar de otra, lo que le confiere a las órdenes Unix una enorme potencia. Ese es el motivo por el que las órdenes Unix sólo realizan estrictamente lo necesario, sin más salida de información adicional que pudiera interferir en la subsiguiente utilización de esta información.

### Uniendo órdenes

Para unir la salida de una orden con la entrada de otra utilizamos el metacaracter de shell '|', poniendo a la izquierda la primera orden que queremos ejecutar y a la derecha la orden que tiene que ejecutarse con la información de la orden anterior.

El carácter de tubería realiza una doble redirección: toma la salida estándar de la orden que hay a su izquierda para pasarla como entrada estándar de la orden que hay a su derecha.

Podemos unir tantas órdenes como nos interese.

Vemos unos ejemplos:

```
$ ps axu | more
```

En este caso, la salida de la orden 'ps' se pasa como entrada a la orden *more* lo que nos permite verla poco a poco. Pero en Unix se pueden hacer cosas más complejas.

Por ejemplo:

```
$ ps axu | wc -l
```

En este caso la salida de *ps* se pasa a la orden '*wc -l*' que cuenta líneas. De esta forma estamos contando el número de líneas que muestra la orden ps, es decir el número de procesos que hay actualmente ejecutándose en el sistema.

Ejercicio:

Contar el número de usuarios que hay conectados en el sistema.

Contar el número de ficheros que hay en el directorio actual.

### tee

La orden tee copia la entrada estándar en la salida estándar y en uno o más ficheros. Se utiliza cuando queremos pasar información a una orden y además almacenarlos resultados intermedios en un fichero; por ejemplo:

```
$ cat /etc/passwd | tee claves | wc -l
```

## Ejercicios

Averiguar cuantos ficheros hay en el directorio actual, sin incluir los ocultos

```
$ ls | wc -l
```

Averiguar cuantos ficheros hay en el directorio actual,incluyendo los ocultos

```
$ ls -a | wc -l
```

Copiar el fichero /etc/hosts en el directorio actual a la vez que lo vemos por la pantalla

```
$ cat /etc/hosts | tee ./hosts
```

Duplicar un disquete, suponiendo que tenemos dos unidades de discos flexibles

```
$ cat /dev/fd0 > /dev/fd1
```

Duplicar un disquete suponiendo que la máquina tiene una sola unidad:

```
$ cat /dev/fd0 > disco.img
$ cat disco.img > /dev/fd0
$ rm disco.img
```

Añadir la línea "192.168.1.101 pci.bez.es" al fichero hosts del directorio personal

```
echo "192.168.1.101 pci.bez.es" >> ~/hosts
```

Guardar la fecha y la lista de usuarios conectados en este momento en un fichero llamado conectados

```
$ date >> conectados
$ who >> conectados
```

Obtener una lista de usuarios conectados ordenada alfabéticamente

```
$ who | sort
```

Averiguar cuantos usuarios distintos hay en el sistema

```
$ cat /etc/passwd | wc -l
```

Mostrar sólo la quinta línea del fichero /etc/passwd

```
head -5 /etc/passws|tail -1
```

## Tratamiento de ficheros de texto

Los sistemas Unix, y Linux en particular disponen de herramientas avanzadas que permiten la manipulación de ficheros de texto para poder extraer información y modificarlos. Esto es realmente importante ya que la mayoría de los ficheros de configuración de un sistema Unix son ficheros de texto que habitualmente tendremos que manipular.

### Expresiones regulares

Una expresión regular define un conjunto de una o más cadenas de caracteres de acuerdo con una reglas. De forma intuitiva, una expresión regular es una plantilla que especifica un conjunto de cadenas de caracteres.

Definición de expresiones regulares

Vamos a describir las reglas principales que definen un expresión regular:

/ delimita una cadena de caracteres

. equivale a uno o ningún carácter . Por ejemplo /host./ equivale a host, hosts, hostsf, etc.

[] define una clase de caracteres. Por ejemplo [bB] representa un carácter que puede ser 'b' o 'B';[a-z] un carácter que puede ser una letra minúscula; [12345] un número del 1 al 5. Si los caracteres van precedidos de ^ entonces se niegan los caracteres, es decir [^A-Z] representa un carácter que NO sea una letra mayúscula.

* representa cero o más ocurrencias del carácter que lo precede. Por ejemplo aw.*z representa cualquier cadena que empiece por aw y termine en z. (Observe que las expresiones regulares son distintas a las plantillas para nombres de ficheros).

^ Al principio de una expresión indica comienzo de línea.

$ Al final de una expresión indica final de cadena.

\ Interpreta el siguiente carácter como carácter y no como una regla de expresión regular. Este es el carácter de escape para proteger caracteres con significado propio.

& Representa un valor tomado por una expresión regular.

| Une dos expresiones regulares mediante un o lógico, es decir se verifica una u otra.

Es importante proteger con el carácter de escape (\) aquéllos caracteres que tengan un significado propio.

Ejemplos

/.*txt/ Una cadena que termine en "txt".

/.*\.txt/ Una cadena que termine en ".txt".

/^Esto/ Una línea que comience por "Esto".

/fin\./ Una cadena que termina en "fin.".

/[a-zA-Z]*/ Cualquier cadena formada por letras.

/[a-z][1-9]*/ Cualquier cadena que empiece por una letra minúscula y el resto sean números.

/y\/o/ Equivale a "y/o".

/[^a-z].*/ Cualquier cadena que no empiece por una letra minúscula.

/(.*)/ La cadena más larga comprendida entre ( y ).

## Uso de expresiones regulares

### grep

La orden grep busca una expresión regular en un fichero mostrando el resultado en la salida estándar.

La orden grep se usa como:

```
$ grep [opciones] expresión_regular [lista_ficheros ...]
```

Consultar la página de manual correspondiente para tener más detalle sobre las opciones disponibles de grep.

Un uso habitual de grep es usar como entrada la salida estándar de otra orden para filtrar la información y obtener sólo aquéllas líneas que nos puedan interesar.

A continuación vemos algunos ejemplos prácticos:

```
$ grep pc[1-9]* /etc/hosts
$ grep procmail /etc/sendmail.cf
$ grep no.* /etc/passwd
$ grep ^n.* /etc/passwd
$ grep false$ /etc/passwd
$ grep no.* /etc/passwd
$ grep '^p\|^n' /etc/passwd
$ grep tcp /etc/inetd.conf
$ grep ^1 /etc/inittab
$ grep "in\." /etc/inetd.conf
$ ps axu | grep nobody
$ ps axu | grep httpd
$ ps axu | grep httpd | wc -l
$ who | grep tty1
$ ls -la | grep \.bashrc
$ ls -la | grep ^-rw
$ ls -la | grep ^d
$ ls -la | grep ^l
```

### egrep

La orden egrep permite realizar un grep en múltiples ficheros. Vea la opción -E de grep.

Por ejemplo, si queremos buscar la palabra "fail" en cualquier fichero del directorio /var/log, pondríamos:

```
$ egrep fail /var/log/*
```

Observamos que pueden aparecer línea de error diciendo que ha intentado buscar en un directorio. Si queremos descartar estas líneas de error pondríamos lo siguiente:

```
$ egrep fail /var/log/* 2>/dev/null
```

## sed

sed es un editor de texto no interactivo. Su principal utilidad es poder editar ficheros de textos desde un programa. Hay que observar que el resultado de la edición se muestra en la salida estándar.

La orden sed dispone de diversas opciones, pero aquí sólo vamos a ver una, la sustitución de un texto por otro.

En general la orden sed se usa como:

```
$ sed orden_sed fichero
```

Donde:

orden es una instrucción de sed. Entre las órdenes está 's' que se utiliza para sustituir texto. En este caso la orden queda como:

```
s/expr_reg_origen/nuevo valor/
```

Si al final añadimos 'g' indicamos que haga una sustitución global en todo el fichero.

Vamos a ver un ejemplo:

Supongamos que tenemos un fichero de texto llamado ftexto con el siguiente contenido:

```
$ cat ftexto
enero, febrero, noviembre, abril
a12, a3, s34, e56, 123,
lunes martes 23 37
```

Veamos los resultado que obtendríamos con diferentes órdenes de sed:

```
$ sed "s/12/zzzzzzz12zzzzzzzz/g" ftexto
enero, febrero, noviembre, abril
azzzzzzz12zzzzzzzz, a3, s34, e56, zzzzzzz12zzzzzzzz3,
lunes martes 23 37
```

Otro ejemplo:

```
$ sed "s/noviembre/diciembre/" ftexto
enero, febrero, diciembre, abril
a12, a3, s34, e56, 123,
lunes martes 23 37
```

También podemos hacer que el texto buscado aparezca en el texto sustituido, usando &. Por ejemplo

```
$ sed "s/[0-9][0-9]/& dos numeros/g" ftexto
enero, febrero, noviembre, abril
a12 dos numeros, a3, s34 dos numeros, e56 dos numeros, 12 dos numeros3,
lunes martes 23 dos numeros 37 dos numeros
```

Como la orden sed escribe sus resultados en la salida estándar, si queremos modificar un fichero tendremos que hacerlo en varios pasos redirigiendo la salida estándar.

Por ejemplo, si queremos que en el nuevo fichero ftexto donde pone lunes ponga domingo tendríamos que poner:

```
$ sed "s/lunes/domingo/" ftexto $gt; ftexto.nuevo$
cp ftexto.nuevo ftexto
$ rm ftexto.nuevo
```

## Otras órdenes útiles para ficheros

### cut

La orden cut corta un trozo de línea de su entrada estándar de acuerdo con las opciones especificadas.

La orden cut se usa como:

```
$ cut opciones fichero
```

y muestra los resultados en la salida estándar.

Las opciones más importantes de esta orden son:

- -c n1-n2 Corta caracteres desde el que ocupa la posición n1 hasta la posición n2.
- -d separador Indica el carácter que actúa como separador
- -f c1 [,c2,...] Indica qué campos queremos cortar. c1 es un número que representa el orden del campo que queremos cortar. Se supone que los campos están separados por el carácter indicado con la opción -d

Ejemplos:

```
   $ cut -d: -f1 /etc/passwd
$ who | cut -c 10-17
$ ls -la | cut c 2-10
$ ls -la | grep ^d | cut -c 2-10
$ cut -d: -f 1,3 /etc/group
$ ps axu | grep root | cut -c 1-6
$ cat /etc/hosts | grep ^[0-9] | cut -c 1-6
$ grep udp /etc/inetd.conf | cut -d: -f 1,2,3,4
$ grep ^[^#] /etc/inittab | cut -d: -f 2,4,5
$ cut -d: -f 1,6 /etc/passwd > lista
```

### spell, aspell

Estas órdenes verifican la ortografía de un fichero de texto respecto a un diccionario y un conjunto de reglas de formación de palabras.

Estas órdenes hay que instalarlas explícitamente en un sistema. Muchos programas, sobre todo editores de texto, utilizan estas órdenes del sistema.

Para más detalles consultar las páginas del manual.

## Permisos y propietarios

Con anterioridad ya habíamos hablado de los permisos y de los propietarios de un fichero o directorio; ahora entramos en más detalles.

Existen órdenes para modificar, tanto los permisos como los propietarios de un fichero. También podremos establecer los criterios sobre los permisos de los ficheros de nueva creación.

### Propiedad

Cualquier sistema operativo multiusuario tiene que proporcionar mecanismos para que cada usuario pueda poseer su propios datos. En el caso de los sistema Unix y Linux en particular a cada fichero o directorio se le puede asignar un usuario propietario y un grupo propietario. La lista de usuarios está en el fichero */etc/passwd* y la lista de grupos está en el fichero */etc/group*. Cada usuario tiene un grupo principal ya además puede pertenecer a cualquier otro grupo.

Combinando el propietario, el grupo de un fichero y los grupos a los que pertenecen los diversos usuario se puede delimitar perfectaemente el acceso a los datos almacenados. Además los sistemas Unix existen ciertos usuarios y grupos estándares que se utilizan para realizar diversas tareas administrativas. En el apartado de administraciñón de usuarios se verá esto con más detalle.

En general, el propietario y el grupo de los ficheros y directorios se heredan del usuario que los crea.

En muchos casos nos interesa modificar el propietario o el grupo de un fichero; para realizar estas acciones Linux (Unix en general) disponen de las órdenes adecuadas.

chown

Para cambiar el propietario de un fichero tenemos que usar la orden *chown* (change owner). La orden chown se utiliza de la siguiente forma:

```
$ chown nuevo_propietario fichero_o_dir
```

La orden chown también se puede utlizar para cambiar también el grupo de un fichero o directorio; esto se hace con la siguientes sintaxis de la orden *chown*:

```
$ chown nuevo_usuario.nuevo_grupo fichero_o_dir
```

o bien

```
$ chown nuevo_usuario:nuevo_grupo fichero_o_dir
```

poniendo el nuevo usuario y el nuevo grupo separados por un punto. De esta forma ahora el fichero o el directorio pertenecen al nuevo usuario y al nuevo grupo. Si omitimos el usuario pero ponemos el punto o los dos puntos sólo se cambia el grupo.

### chgrp

La orden chgrp cambia el grupo de un fichero o directorio. Se utiliza como

```
$ chgrp nuevo_grupo fichero_o_dir
```

Esta orden tiene la misma funcionalidad que

```
$ chown :nuevo_grupo fichero_o_dir
```

## Permisos

### chmod

La orden chmod cambia los permisos de un fichero o directorio. Se usa como:

```
$ chmod [opciones] permiso fichero
```

Donde permiso se puede especificar de dos formas distintas:

a) En octal, donde un 0 representa un permiso denegado y un 1 un permiso concedido. El número octal está formado por tres dígitos; el de más a la izquierda representa representa los permisos del usuario propietario del fichero o (directorio). El dígito central representa los permisos del grupo propietario del fichero y el dígito de la derecha representa los permisos que se conceden al resto de los usuarios del sistema. Cada uno de los dígitos octales se descompone en tres dígitos binarios donde de izquierda a derecha significan lectura (r), escritura (w) y ejecución (x).

Por ejemplo: 764 indica

```
Propietario 7 = 111 = rwx
Grupo 6 = 110 = rw-
Otros 4 = 100 = r--
```

b) En formato simbólico, de la forma "d+|-p", donde "d" es el destinatario del permiso y puede ser "u" (usuario propietario), "g" (grupo propietario) y "o" (otros, el resto de los usuarios); también se puede poner "a" que significa todos (all). Un signo "+" activa un permiso y un signo "-" lo desactiva. El siguiente carácter "p" representa el permiso que puede ser r (lectura), w (escritura) y x (ejecución) o una combinación de ellos.

Por supuesto, el fichero indica quien recibe los permisos.

Realizar las siguentes operaciones en su sistema linux y ver como se modifican los permisos:

```
$ touch mifi
$ chmod ug+wr mifi
$ ls -la mi*
$ chmod o-wrx mifi
```

```
$ ls -la mifi
$ chmod 700 mifi
$ ls -la
$ chmod 731 mifi
$ ls -la
```

y podremos ver como van cambiando los permisos del fichero con las sucesivas ejecuciones de la orden chmod.

Si a la orden chmod le añadimos ls opción -R modifica los permisos de forma recursiva, es decir incluyendo toda la rama del árbol de directorios a partir de donde se ejecuta la orden.

Hay que observar que el comportamiento de los distintos permisos es distinto para ficheros y directorios:

- El permiso de lectura en un fichero implica poderlo leer mientras que en un directorio significa mostrar su contenido.
- El permiso de escritura en un fichero indica poderlo borrar o modificar mientras que para un directorio indica crear ficheros en él o borrar los existentes.
- El permiso de ejecución para un fichero implica que se pueda ejecutar como programa, mientras que un directorio significa poder entrar en él.

### Los bits SUID y SGID

Existen otros dos permisos, propios de los ficheros ejecutables que modifican las características de los procesos generados al ejecutarlos si estos permisos están activos. Antes de entrar en más detalles vamos a aclarar dos conceptos sobre los procesos que se lanzan:

- usuario real: el usuario que está conectado y que lanza el proceso.
- usuario efectivo: el usuario que es propietario de un proceso, y cuyos permisos le son aplicables.

Esto es importante porque algunos programas necesitan ejecutarse como un usuario distinto al usuario que lo lanza. Por ejemplo, el fichero /etc/passwd es propiedad del usuario root y del grupo root y tienen los permisos *rw-rw-r--* o *rw-r--r--*, lo que quiere decir que cualquiera puede leerlo pero sólo el root puede escribir en él. Pero un usuario para poder modificar su clave de acceso tiene que escribir en el fichero. Pero claro, al ejecutar el programa passwd, este proceso es propiedad del usuario que lo lanza y en consecuencia no puede modificarlo. Entonces ¿cómo podemos solucionar esto? Pues haciendo que el proceso generado por la orden passwd sea propiedad del root y activando el bit SUID. De esta forma cuando la orden passwd genera un proceso se genera como un proceso que es propiedad del propietario del fichero, y no propiedad del usuario que lo lanza.

Hay que limitar en la medida de lo posible los fichros ejecutables con el bir SUID activo, sobre todo si los ficheros son propiedad del root, ya que pueden originar problemas de seguridad, que alguien consiga acceso de root de forma indebida.

El permiso SGID es igual el SUID salvo que se aplica al grupo en lugar de la usuario. No son incompatibles.

Los bits SUID y SGISD los podemos asignar como:

```
$ chmod u+s fichero
$ chmod g+s fichero
```

### El sticky bit

Este es un bit que tiene un significado para los directorios. Cuando este bit está activo, hace que un usuario sólo pueda borrar los ficheros que son de su propiedad en dicho directorio. Esto es particularmente útil en el directorio */tmp*.

El sticky bit se activa como:

```
$ chmod +t directorio
```

Permisos preseterminados para nuevos ficheros:s: umask

A un usuario le puede interesar definir cuales son los permisos que se asignan a los ficheros que se crean nuevos. Resulta útil si queremos evitar que se puedan crear ficheros con permisos inadecuados y permitan la lectura o modificación por usuarios indebidamente.

La orden umask es la que se encarga de establecer una máscara que prefija unos determinados permisos para cada nuevo fichero. La forma de uso de umask es la siguiente:

```
umask 0XXX
```

donde cada X representa un número octal, en el cual, daad su expresión en binario, cada uno representa un permiso no asignado y un 0 representa un permiso asignado, alrevés que *chmod*:

Ejemplo:

```
$ umask 0011
$touch borrar1
$ ls -la bo*
drwxrwxr-x    4 pfabrega pfabrega     4096 nov 23 20:14 .
drwx--x--x   20 pfabrega pfabrega     4096 nov 23 19:26 ..
-rw-rw-rw-    1 pfabrega pfabrega        0 nov 23 20:14 borrar1
$ umask 0122
$ touch borrar2
$ ls -la bo*
drwxrwxr-x    4 pfabrega pfabrega     4096 nov 23 20:15 .
drwx--x--x   20 pfabrega pfabrega     4096 nov 23 19:26 ..
-rw-rw-rw-    1 pfabrega pfabrega        0 nov 23 20:14 borrar1
-rw-r--r--    1 pfabrega pfabrega        0 nov 23 20:15 borrar2
```

## Ejercicios:

Vemos ahora un lista de ejercicios para comprobar como actúan las últimas órdenes.

Ejercicio 1

Crear un fichero llamado borrar:

```
$ touch borrar
```

Verificar los permisos y los propietarios y grupo.

```
$ ls -la bo*
```

Asignar los permisos necesarios para que podamos borrar el fichero si lo cambiamos de propietario.

```
$ chmod g+w borrar
```

Pasar la propiedad del fichero borrar a root:

```
$ chown root borrar
```

Verificar que se ha realizado los cambios:

```
$ ls -la bo*
```

Intentar ponerle al fichero los todos los permisos a todo el mundo:

```
$ chmod a+wrx borrar
```

Borrar el fichero:

```
$ rm borrar
```

Ejercicio 2

Crear un fichero llamado eliminar

```
$ touch eliminar
```

Verificar los permisos del fichero

```
$ ls -la el*
```

Asignar los permisos necesarios para poderlo borrar si cambiamos el grupo y propietario.

```
$ chmod o+wr eliminar
```

Cambiar el fichero eliminar al usuario root y al grupo root.

*$ chown root.root eliminar*

Verificar si se han realizado los cambios:

```
$ ls -la elim*
```

Asignar permisos de ejecución a eliminar.

```
$ rm eliminar
```

Ejercicio 3
Crear un fichero llamado pr1

```
$ touch pr1
```

Asignarle el permiso de ejecución para todos los usuarios:

```
$ chmod a+x pr1
```

Añadir una línea al fichero que sea "*sleep 100*":

```
$ echo "sleep 100" >> pr1
```

Ejecutar pr1 en segundo plano:
*$ ./pr1 &*
Ver quien es el propietario del proceso generado:

```
$ ps axu | grep pr1
```

Copiar el fichero pr1 en otro llamado pr2

```
$ cp pr1 pr2
```

Asignarle el bit SUID a pr2

```
$ chmod +s pr2
```

Verificar que se ha asignado el permiso

```
$ ls -la pr*
```

Abrir una nueva sesion de trabajo
*(telnet o ALT-F2) login: practicas*
Lanzar ambos procesos en segundo plano

```
$ ./pr1 &; ./pr2 &
```

Comprobar los propietarios de ambos procesos

```
$ ps axu | grep "pr[12]"
```

# Shell

La shell es el intérprete de órdenes que utiliza Linux, bueno Unix en general. No hay una sola shell, sino que cada usuario puede elegir la que quiera en cada momento. Además las shells funcionan como lenguajes de programación de alto nivel, lo que se estuaciará en un capítulo posterior.

En principio hay dos familias de shell, las shell de Bourne y las shell C. Dentro de cada una de ellas hay diferentes vesiones, como por ejemplo ksh (Korn shell) o bash (Bourne Again Shell),. Nosotros en Linux vamos a usar esta última, aunque gran parte de las cosas son aplicables a otras shell.

## Definiciones

### Variables de shell

Una variable de shell es una zona de almacenamiento de información con un nombre que la identifica. Para no liarnos mucho, es similar a un variable en lenguaje C. Podemos asignar un valor a una variable mediante el operador "=". Si la variable no existe el operador "=" la crea. No se debe dejar espacio entre el nombre de la variable y el operador de asignación (=).

Para usar el valor de una variable es necesario anteponerle el símbolo "$".

Ejemplos:

```
$ AA=Hola que tal
$ echo AA
$ echo $AA
```

Como norma de estilo, se suelen usar letras mayúscular para definir los nombres de las variables.

### Variables de entorno

Son variables que tienen un significado propio para la shell o algún otro programa. Entre ellas podemos citar:

PATH Indica la ruta de búsqueda de programas ejecutables. Está constituida por una lista de directorios separados por ":". El directorio actual, de forma predeterminada, no viene incluida en PATH.

PS1 Especifica el indicador del sistema. Lo habitual es que PS1 sea "$" para usuarios normales y "#" para root.

PS2 Especifica el indicador secundario del sistema. Aparece cuando no se ha completado una orden.

LANG Especifica el lenguaje que se aplica al usuario. Para español se utiliza "es".

LC_ALL Contiene el idioma y se utoiliza para usar los valores locales como mensajes del sistema, símbolo monetario, formato de fecha, formato de números decimales y otras características.

TERM Indica el tipo de teminal que se está utilizando. Por ejemplo, si estamos en la consola el valor de TERM será "linux", para usar las carácterísticas del teclado. Si entramos por telnet desde w9x entonces probablemente tengamos que poner vt100.

EDITOR Especifica el editor por omisión del sistema. Lo habitual en los sistema Unix es que el editor por omisión sea "vi".

DISPLAY Especifica qué equipo muestra la salida que se efectúa en modo gráfico. Ese equipo deberá tener un servidor gráfico.

LD_LIBRARY_PATH Esta variable se utiliza para definir rutas alternativas de búsqueda de para bibliotecas de funciones del sistema.

PWD Contiene el directorio de trabajo efectivo.

Con la orden *env* (environment) podemos comprobar el valor de las variables de entorno del sistema. Para modificarlas basta asignarle un nuevo valor.

Ejemplo:

```
$ env
PWD=/home/usuario/public_html/docs/shell
LTDL_LIBRARY_PATH=/home/usuario/.kde/lib:/usr/lib
HOSTNAME=www.bdat.com
LD_LIBRARY_PATH=/home/usuario/.kde/lib:/usr/lib
QTDIR=/usr/lib/qt-2.2.2
CLASSPATH=./:/usr/lib/jdk118/lib/classes.zip
LESSOPEN=|/usr/bin/lesspipe.sh %s
PS1=[\u@\h\W]$
KDEDIR=/usr
BROWSER=/usr/bin/netscape
USER=usuario
MACHTYPE=i386-redhat-linux-gnu
LC_ALL=es_ES
MAIL=/var/spool/mail/usuario
EDITOR=joe
LANG=es
COLORTERM=
DISPLAY=:0
LOGNAME=usuario
SHLVL=4
LC_CTYPE=iso-8859-1
SESSION_MANAGER=local/www.bdat.com:/tmp/.ICE-unix/965
SHELL=/bin/bash
HOSTTYPE=i386
OSTYPE=linux-gnu
HISTSIZE=100
HOME=/home/usuario
TERM=xterm
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/lib/jdk118/bin:./:/sbin:/usr/sbir
GROFF_TYPESETTER=latin1
LESSCHARSET=latin1
```

## Ficheros ejecutables

Un fichero ejecutable es un fichero que tenga asignado el permiso de ejecución.

Los ficheros ejecutables pueden ser de dos tipos:

- Binarios, cuando son resultado de la compilación de un programa, por ejemplo en C.
- Programas de shell, llamados scripts o guiones. Son ficheros de texto que contienen órdenes de shell y son interpretados por una shell.

## Shell y subshell

Podemos abrir una nueva shell simplemente ejecutando el fichero binario que contiene la shell.

También, cuando ejecutamos un script se abre una nueva shell que es la encargada de ir interpretando las diferentes órdenes y de mantener el valor de las variables definidas. Esta subshell tiene como tiempo de vida el tiempo de ejecución del script. Además esta subshell hereda el valor de parte de las variables de entorno, pero en su propio espacio de memoria, por lo que las modificaciones de cualquier variable de la nueva shell no tendrá repercusión en la shell padre.

## Cerrar una shell

Para cerrar una shell usamos la orden exit. Esta orden termina la ejecución de la shell y vuelve a la shell padre. Si ejecutamos la orden exit sobre la shell inicial que abrimos al entrar al sistema (llamada login shell) entonces terminamos la sesión de trabajo.

## Variables exportadas

En el apartado anterior, veíamos que una subshell hereda parte de las variables definidas en la la shell padre. Para que las shell hijas de una dterminada shell hereden una variable es necesario indicarlo explícitamente con la orden export. Por ejemplo, si queremos que el valor de la variable EDITOR pase a las subshell que sean hijas de la shell activa, tendremos que poner

```
$ export EDITOR
```

En la shell bash, podemos realizar en una sola operación la asignación de una valor y exportarla, por ejemplo

```
$ export EDITOR=vi
```

No todas las shell permiten esta operación

Ejemplos:

Asignar la palabra contenido a una variable llamada AA

```
$ AA=contenido
```

Mostrar el contenido de la variable en pantalla.

```
$ echo $AA
```

Abrir una shell hija de la anterior.

```
$ /bin/bash
```

Mostrar de nuevo el contenido de la variable

```
$ echo $AA
```

Salir de la subshell activa

```
$ exit
```

Mostrar de nuevo el contenido de la variable

```
$ echo $AA
```

Exportar la variable

```
$ export AA
```

Abrir una shell hija de la anterior.

```
$ /bin/bash
```

Mostrar de nuevo el contenido de la variable

```
$ echo $AA
```

## Las comillas en la shell

En las operaciones con la shell distinguimos tres tipos de comillas con distintas funcionalidades: las comillas dobles, las comillas simples y la comilla invertica, el acento grave francés. A continuación describimos las funcdiones:

' Engloban un literal. La shell no trata de interpretar nada de lo que haya comprendido entre estas comillas.

" La shell expande las variables que haya especificadas sustituyendo su nombre por el contenido. Para imprimir un $ es necesario protegerlo con una \.

\‘{ } La shell intenta ejecutar lo que haya comprendido entre estas comillas.

Ejemplo:

Asignamos una valor a una variable:

```
$ AA="ls -la"
```

y observamos la diferencia entre:

```
echo '$AA'
echo "$AA"
```

y

```
echo \‘{ }$AA\‘{ }
```

## Personalización de la shell

Cada vez que se inicia una shell, se lee un fichero de configuración. No es un fichero complejo, es simplemente un fichero con ódenes que se se ejecutan automáticamente cada vez que se inicia una nueva shell.

Diferentes shell utilizan diferentes ficheros de configuración. Las shell C suelen llamar a este fichero .login. Las shell de Bourne suelen llamar a este fichero *.profile*. En la shell bash, además del *.profile*, tenemos también el fichero .bashrc.

En este fichero vamos a ejecutar las órdenes y asiganar valores a variables necesarios para adaptar la shell a nuestras necesidades. En general, todo aquéllo que queramos que se ejecute cada vez que entremos al sistema. Por ejemplo, nos puede interesar añadir a este fichero una línea como:

```
$ export LANG=es
```

para hacer que nuestro idioma predeterminado sea el español.

También podemos poner . (directorio efectivo) en la ruta de búsqueda de programas ejecutables:

```
$ export PATH=$PATH:.
```

La shell bash también usa el fichero *.inputrc* para configurar el teclado.

## El editor vi

### Intoducción

El editor vi (visual) es el programa de edición común a todos los sistemas Unix. Aún cuando es posible utilizar otros editores, es necesario conocer su funcionamiento en determinadas situaciones críticas.

Modos de trabajo

El editor vi tiene dos modos de trabajo:

- Modo inserción: En este modo se puede introducir texto en el fichero que estamos editando. Para salir del modo inserción tenemos que pulsar la tecla Esc.
- Modo orden: En este modo *vi* acepta órdenes. Las órdenes son letras con algún posible arguemento.

### Terminales

Para el correcto funcionamiento del *vi* tenemos que asegurarnos que las características de la terminal son las adecuadas. Por ejemplo, si trabajamos con un terminal vt100 deberíamos poner:

```
$ TERM=vt100
$ export TERM
```

La línea inferior de la pantalla la usa vi como línea de estado, para escribir algunos mensajes y para escribir las órdenes.

### Salir de vi

Para salir de vi hay que estar en modo orden, por lo que se tendrá que pulsar Esc.

orden acción

ZZ sale de vi grabando el fichero

:q! sale de vi sin grabar los cambios

:wq sale de vi grabando los cambios

### Introducir texto (modo inserción)

orden acción

i inserta texto en la posición del cursor

a inserta texto después de la posición del cursor

A inserta texto al final de la línea

o Crea una nueva línea bajo la actual

O Crea una nueva línea sobre la actual

### Borrar

orden acción

x borra el carácter sobre el cursor

d0 borra hasta princio de línea

dw borra hasta el final de la palabra.

dnw borra hasta el final de la palabra n

db borra hasta el principio de la palabra

dd borr la línea actual

### Desplazamientos

orden acción

->,espacio espacio a la derecha

<-,h espacio a la izquierda

w palabra a la derecha

b palabra a la izquierda

$ fin de línea

0 principio de línea

return línea siguiente

j línea de abajo

k línea de arriba

### Búsquedas y sustituciones

orden acción

/expreg busca expreg hacia adelante

?expreg busca expreg hacia atrás

/ repite la última búsqueda hacia adelante

? repite la última búsqueda hacia atrás

s/buscado/sustitución[/g] sustituye la primera aparición de la palabra buscado reemplazándola por la parabra sustitición. Si añadimos /g al final, la sustitución es global en todo el documento.

### Otras órdenes

orden acción

H principio de la pantalla

M mitad de la pantalla

L final de la pantalla

nG A la línea n

( principio de frase

) fin de frase

{ principio de párrafo

} fin de párrafo

r sustituye el carácter del cursor

R sustituye caracteres

D borra hasta el final de línea

:set number numera las líneas

:set ai establece el sangrado automático

:!orden ejecuta el orden de shell

:w graba el fichero sin salir de vi.

## Ejecución y agrupación de órdenes

Una vez vistas gran parte de las órdenes de usuario, pasamos a ver una de las principales características de un sistema Unix que es la facilidad para agrupar órdenes de distintas formas para realizar tareas complejas. Por un lado en la propia línea de órdenes se pueden especificar órdenes y unir su ejecución de diversas formas.

Hasta ahora hemos visto como combinar órdenes con tubería, como redirigir las salida y como gestionar procesos en primer y segundo planos. Ahora vamos a ver otra serie de mecanismos para ejecutar órdenes y programas y verificar el estado de conclusión de la orden.

### Código de terminación de una orden

Cuando una orden termina le devuelve al sistema un código de finalización, un valor cero en caso de terminar correctamente o un valor uno si se ha producido un error. Este valor se almacena en la variable $?.

Por ejemplo

```
[pfabrega@port /home]$ ls -la /home
total 32
drwxr-xr-x    5 root     root         4096 sep 20 14:16 .
drwxr-xr-x   18 root     root         4096 sep 20 03:23 ..
drwxr-xr-x    4 root     root         4096 sep 28 21:16 httpd
drwxr-xr-x    2 root     root        16384 sep 19 18:25 lost+found
drwx--x--x   20 pfabrega pfabrega     4096 nov 24 19:07 pfabrega
[pfabrega@port /home]$ echo $?
0
[pfabrega@port /home]$ ls -la asdfg
ls: asdfg: No existe el fichero o el directorio
[pfabrega@port /home]$ echo $?
1
[pfabrega@port /home]$
```

Podemos observar como en la primera ejecución *ls -la* devuelve un valor 0 de terminación correcta y en la segunda devuelve un error y n código 1.

## Ejecución consecutiva

Podemos agrupar varias órdenes en una misma línea de ordenes separándolas por ";"

La agrupación de órdenes separadas por ";" es útil cuando tenemos que repetir una misma secuencia de órdenes varias veces.

La ejecución de cada una de las órdenes se realiza cuando ha concluido la anterior, e independiente de que el resultado haya sido correcto o no.

Esto nos puede resultar útil para demorar la ejecución de una o varias órdenes un determinado tiempo, por ejemplo

```
$ sleep 300 ; ps axu
```

y la orden *ps axu* se ejecutaría a los 300 segundos.

## Ejecución condicional

Otra situación algo más elaborada que la anterior es ejecutar una orden condicionada a la terminación correcta o no de una orden previa.

Esta funcionalidad nos la proporcionan los operadores "*&&*" y "||".

### Operador &&

El primer operador, "*&&* " separa dos órdenes de forma que la que tiene a la derecha sólo se ejecuta cuando la de la izquierda termina correctamente, es decir

```
orden1 && orden2
```

*orden2* sólo se ejecutará si *orden1* terminó sin ningún error.

Por ejemplo, queremos ejecutar la orden *cat fichero* sólo si existe *fichero;* entonces tendremos que buscar una orden que termine con un error si no existe *fichero*, por ejemplo *ls fichero* y condicionar la ejecución de *cat fichero* a esta:

$ ls fichero && cat fichero

Otro ejemplo, para compilar los controladores de dispositivos de linux e instalarlos, lo podemos hacer como:

make module && make modules_install

es decir instalará los controladores sólo si ha conseguido compilarlos correctamente.

### Operador ||

El segundo operador, "|| " tiene un comportamiento similar al anterior, separa dos órdenes de forma que la que tiene a la derecha sólo se ejecuta cuando la de la izquierda termina incorrectamente, es decir

```
orden1 || orden2
```

*orden2* sólo se ejecutará si *orden1* terminó con algún error.

Por ejemplo si no existe fichero queremos crearlo vacía y si existe no hacemos nada. Igual que en el ejemplo anterior, buscamos una orden que termine con un error si no existe fichero y condicionamos la ejecución de la orden touch fichero al error de la orden previa:

```
$ ls fichero || touch fichero
```

También, al igual que en el ejempo anterior podríamos hacer que si el proceso *make modules* falla, se borraran todos los ficheros temporales que se crean:

```
make modules || rm -r *.o
```

## Ejecución simultánea

Otra posibilidad de ejecución también posible es lanzar varios procesos simutáneamente en segundo plano; basta escribir uno a continuación de otro en la línea de órdenes separados por "&". Este es el símbolo que se utiliza para indicar que el proceso se tiene que ejecutar en segundo plano, pero también actúa como separador para la ejecución de distintas órdenes.

por ejemplo :

```
[pfabrega@port pfabrega]$ sleep 10 & sleep 20 & sleep 15 &
[pfabrega@port pfabrega]$ ps axu
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.2  1408  540 ?        S    22:19   0:04 init [3]
...
pfabrega  1263  0.3  0.2  1624  516 pts/4    S    23:24   0:00 sleep 10
pfabrega  1264  0.0  0.2  1624  516 pts/4    S    23:24   0:00 sleep 20
pfabrega  1265  0.0  0.2  1624  516 pts/4    S    23:24   0:00 sleep 15
pfabrega  1266  0.0  0.3  2732  848 pts/4    R    23:24   0:00 ps axu
```

## Agrupando con paréntesis

Podemos organizar la ejecución de varias órdenes agrupándolas convenientemente mediante paréntesis para modificar el orden predeterminado de ejecución. En primer lugar actúan los ; después los & y la ejecución es de izquierda a derecha. Para las ejecuciones condicionales se tiene en cuenta el valor de la variable *$?* que se fija por la última orden que se ejecuta.

Para alterar esta forma de ejecución podemos utilizar los paréntesis. En realidad los paréntesis fuerzan una nueva subshell para ejecutar las órdenes correspondientes.

Esta característica puede ser interesante en diferentes situaciones:

Quemos enviar una secuencia de órdene s a segundo pláno

```
(mkdir copiaseg; cp -r ./original/* ./copiaseg; rm -r ./original~; rm -r ./original.tmp
```

## Resultado de la ejecución de una orden

Es habitual necesitar almacenar el resultado de la ejecución de una orden en una variable en lugar de que se dirija a la salida estándar o simplemente ejecutar como orden el resultado de otra orden.

### Comillas invertidas \‘{ }\‘{ }

Las comillas invertidas consideran una orden lo que tengan dentro y lo ejecutan devolviendo el resultado como líneas de texto a la shell. Por ejemplo:

```
$ A="ls /bin"
$ ‘echo $A‘
arch        consolechars   ed          igawk     mount          rpm        tar
ash         cp             egrep       ipcalc    mt             rvi        tcsh
ash.static  cpio           ex          kill      mv             rview      touch
awk         csh            false       ln        netstat        sed        true
basename    date           fgrep       loadkeys  nice           setserial  umount
bash        dd             gawk        login     nisdomainname  sfxload    uname
bash2       df             gawk-3.0.6  ls        ping           sh         usleep
bsh         dmesg          grep        mail      ps             sleep      vi
cat         dnsdomainname  gtar        mkdir     pwd            sort       view
chgrp       doexec         gunzip      mknod     red            stty       ypdomainname
chmod       domainname     gzip        mktemp    rm             su         zcat
chown       echo           hostname    more      rmdir          sync
```

También podríamos haber puesto:

```
$ A="ls /bin"
$ B=‘$A‘
$ echo $B
arch ash ash.static awk basename bash bash2 bsh cat chgrp chmod chown consolechars cp c
```

### El operador $()

La shell bash proporciona el operador $() similar a las comillas invertidas. Ejecuta como orden los que haya entre paréntesis y devuelve su resultado. El mecanismo de funcionamiento es idéntico, con la ventaja de poder anidar operadores.

## Programas de shell

Además de las anteriores posibilidades también se pueden agrupar una serie de órdenes en un fichero de texto que se ejecutarán consecutivamente siguiendo el flujo determinado por órdenes de control similares a cualquier lenguaje de programación. Estos ficheros se conocen como scripts, guiones o simplemente programas de shell. A las órdenes agrupadas en ficheros también se le aplican todas las características descritas anteriormente. No olvidemos que para un sistema unix, una línea leída de un fichero es idéntica a una línea leída desde el teclado, una serie de caracteres terminado por un carácter de retorno de carro.

Cualquier forma de ejecución que se pueda dar en la línea de órdenes también se puede incluir en un fichero de texto, con lo que facilitamos su repetición. Y si por último añadimos las estructuras que controlan el flujo de ejecución y ciertas condiciones lógicas, tenemos un perfecto lenguaje de programación para administrar el sistema con toda facilidad. Un administrador que sabe cual es su trabajo cotidiano, realizar copias de seguridad, dar de alta o baja usuarios, comprobar que los servicios están activos, analizar log de incidencias, configurar cortafuegos, lanzar o parar servicios, modificar configuraciones, etc., normalmente se creará sus script personalizados. Algunos los utilizará cuando sea necesario y para otros programará el sistema para que se ejecuten periódicamente.

La programación en shell es imprescindible para poder administrar un sistema Unix de forma cómoda y eficiente.

Vemos un primer ejemplo:

```
#!/bin/bash                    echo Hola Mundo
```

Puestas estas dos línea en un fichero de texto con permiso de ejecución, al ejecutarlo escribiría en pantalla "Hola Mundo". La primera línea, como veremos con posterioridad, indica qué shell es la que interpreta el programa de shell.

### subshell

Para la ejecución de programas de shell, la shell se encarga de interpretar unas órdenes en unos casos o de lanzar el programa adecuado en otros casos. En general, cuando lanzamos la ejecución de un conjunto de órdenes agrupadas en un programa de shell, se abre una nueva shell (subshell hija de la anterior) que es la encargada de interpretar las órdenes del fichero. Una vez concluida la ejecución esta subshell muere y volvemos a la shell inicial. Esto es importante tenerlo presente para saber el comportamiento de los programas. Por ejemplo, los cambios hechos en las variables de shell dentro de un programa no se conservan una vez concluida la ejecución.

Vamos a ilustrar este comportamiento con nuestro primer ejemplo de programa de shell:

Creamos un programa en un fichero de texto, lo ejecutamos, comprobamos que se crea una nueva shell y que los cambios en las variables hechos dentro del programa no se mantienen una vez concluido. Editamos un fichero llamado "pruebashell" con el siguiente contenido:

```
echo "******** GUION ********"
echo "el valor previo de VAR es ** $VAR **"
VAR="valor asignado dentro del guion"
echo "Ahora VAR vale ** $VAR **"
ps
echo "******** FIN DEL GUION ********"
```

Con este guion mostramos el valor previo de una variable llamada VAR, le asignamos un valor nuevo y también los mostramos. Para verificar que se lanza una nueva shell mostramos la lista de procesos con la orden *ps*.

Una vez editado el fichero tendremos que asignarle el permiso de ejecución

```
$ chmod u+x pruebashell
```

después asignamos un una valor a la variable VAR para comprobar como cambia. Además tendremos que exportarla par que la shell hija pueda heredarla:

```
$ export VAR="valor previo"
```

Ahora mostramos la lista de procesos para ver cuantas shell tenemos abiertas:

```
$ ps
o bien
$ ps |wc -l
```

y a continuación ejecutamos el guion "pruebashell":

```
$ ./pruebashell
```

y volvemos a mostrar el contenido de la variable:

```
$ echo $VAR
```

Podremos observar como aparece una shell más. Si la variable VAR está exportada veremos como muestra el valor que asignamos antes de ejecutar el guion y como muestra el que le asignamos dentro del guion. Y al final, al mostrar la variable VAR, observamos como nos muestra el valor que tenía antes de ejecutar el guion; el guion no ha modificado la variable.

Este mismo comportamiento se puede aplicar a la orden cd. Veamos el siguiente ejemplo, un simple script que cambia de directorio.

Editamos el fichero llamado "cambia" con el siguiente contenido:

```
echo "cambiando de directorio"
cd /tmp
echo "estamos en:"
pwd
```

Es decir, el script simplemente cambia al directorio /tmp.

Una vez editado le asignomos el permiso de ejecución

```
$ chmod u+x cambia
```

Ahora mostramos nuestro directorio activo

```
$ pwd
```

ejecutamos el script

```
$ ./cambia
```

y volvemos a comproba nuesto directorio activo

```
$ pwd
```

y observamos como estamos situados en el mismo directorio que antes de la ejecución del guion.

¿Por qué ocurre todo esto?, Porque todos los cambios se realizan en la subshell que ha interpretado el guion.

## Comentarios y continuaciones de línea

Para añadir un comentario a un programa de shell se utiliza el carácter #. Cuando la shell interprete el programa ignorará todo lo que haya desde este carácter hasta el final de la línea.

Por otro lado si nos vemos obligados a partir un línea en dos o más, tendremos que finalizar cada línea de texto inconclusa con el carácter \. De esta forma la shell las verá todas ellas como si se tratara de una única línea.

## Parámetros posicionales

En un programa de shell definen unas variables especiales, identificadas por números, que toman los valores de los argumentos que se indican en la línea de órdenes al ejecutarlo. Tras el nombre de un script se pueden añadir valores, cadenas de texto o números separados por espacios, es decir, parámetros posicionales del programa de shell, a los que se puede acceder utilizando estas variables.

La variable *$0* contiene el parámetro 0 que es el nombre del programa de shell.

Las variables *$1, $2, $3, $4, ...* hacen referencia a los argumentos primero, segundo, tercero, cuarto, ... que se le hayan pasado al programa en el momento de la llamada de ejecución.

Por ejemplo, si tenemos un programa de shell llamado *parametros* con el siguiente contenido:

```
echo $0
echo $1
echo $2
echo $3
```

al ejecutarlo

```
$ parametros primero segundo tercero
prametros
primero
segundo
tercero
```

### Modificación de los parámetros posicionales

Durante la ejecución de un programa de shell podría interesarnos modificar el valor de los parámetros posicionales. Esto no lo podemos hacer directamente, las variables 1, 2, ... no están definidas como tales. Para realizar estos cambios tenemos que utilizar la orden set. Esta orden asigna los valores de los parámetros posicionales a la shell activa de la misma forma que se hace en la línea de órdenes al ejecutara un programa; hay que tener en cuenta que no los asigna individualmente, sino en conjunto.

Por ejemplo

```
$ set primero segundo tercero
$ echo $1
primero
$ echo $2
segundo
$ echo $3
tercero
```

### La sentencia shift

La sentencia shift efectúa un desplazamiento de los parámetros posicionales hacia la izquierda un número especificado de posiciones. La sintaxis para la sentencia shift es:

```
$ shift n
```

donde *n* es el número de posiciones a desplazar. El valor predeterminado para *n* es 1. Hay que observar que al desplazar los parámetros hacia la izquierda de pierden los primeros valores, tantos como hayamos desplazado, al superponerse los que tiene a la derecha.

Por ejemplo:

```
$ set uno dos tres cuatro
$ echo $1
uno
$ shift
$ echo $1
dos
$ shift
$ echo $1
tres
$ shift
$ echo $1
cuatro
```

### Operador {}

Hemos visto la forma de acceder a los diez primeros parámetros posicionales, pero para acceder a parámetros de más de dos dígitos tendremos que usar una pareja { } para englobar el número.

```
$ echo ${10}
$echo ${12}
```

El operador { } también se usa para delimitar el nombre de las variables si se quiere utilizarla incluida dentro de un texto sin separaciones:

```
$DIR=principal
$ DIRUNO=directorio
$ UNO=subdirectorio
$ echo $DIRUNO
directorio
$ echo ${DIR}UNO
principalUNO
```

## Variables predefinidas

Además de las variables de shell propias del entorno, las definidas por el usuario y los parámetros posicionales en un shell existen otra serie de variables cuyo nombre está formado por un carácter especial, precedido por el habitual símbolo $.

### Variable $*

* La variable $* contiene una cadena de caracteres con todos los parámetros posicionales de la shell activa excepto el nombre del programa de shell. Cuando se utiliza entre comillas dobles se expande a una sola cadena y cada uno de los componentes está separado de los otros por el valor del carácter separador del sistema indicado en la variable IFS. Es decir si IFS tiene un valor "s" entonces "$*" es equivalente a "$1s$2s...". Si IFS no está definida, los parámetros se separan por espacios en blanco. Si IFS está definida pero tiene un contenido nulo los parámetros se unen sin separación.

## Variable $@

@ La variable $@ contiene una cadena de caracteres con todos los parámetros posicionales de la shell activa excepto el nombre del programa de shell. La diferencia con $* se produce cuando se expande entre comillas dobles; $@ entre comillas dobles se expande en tantas cadenas de caracteres como parámetros posicionales haya. Es decir "$@" equivale a "$1" "$2" ...

## Variable $#

# Contiene el número de parámetros posicionales excluido el nombre del probrama de shell. Se suele utilizar en un guion de shell para verificar que el número de argumentos es el correcto.

### Variable $?

? Contiene el estado de ejecución de la última orden, 1 para una terminación con error o 0 para una terminación correcta. Se utiliza de forma interna por los operadores || y && que vimos con anterioridad, se utilizar por la orden test que vermos más adelante y también la podremos usar explícitamente.

### Variable $$

$ contiene el PID de la shell. En un subshell obtenida por una ejecución con (), se expande al PID de la shell actual, no al de la subshell. Se puede utilizar para crear ficheros con nombre único, por ejemplo $$.tmp, para datos temporales.

### Variable $!

! Contiene el PID de la orden más recientemente ejecutada en segundo plano. Esta variable no puede ayudar a controlar desde un guion de shell los diferentes procesos que hayamos lanzado en segundo plano.

Ejemplos

Vemos algunos ejemplos a continuación:

```
$ set a b c d e
$ echo $#
5
$ echo $*
a b c d e
$ set "a      b" c d
$ echo $#
3
$ echo $*
a     b c d
```

Otro ejemplo, si tenemos el script llamado *ejvar1* con el siguiente contenido

```
ps
echo " el PID es $$"
```

al ejecutarlo

```
 $ ./ejvar1
  PID TTY          TIME CMD
  930 pts/3    00:00:00 bash
 1011 pts/3    00:00:00 bash
 1012 pts/3    00:00:00 ps
el PID es 1011
```

y vemos como muestra el PID de la shell que ejecuta el script.

Como el PID del prodceso es único en el sistema, este valor puede utilizarse para construir nombres de ficheros temporales únicos para un proceso. Estos ficheros normalmente se suelen situar en el directorio temporal /tmp.

Por ejemplo:

```
miproctemp=/tmp/miproc.$$
. . . .
```

```
rm -f $miproctemp
```

## Uso de valores predeterminados de variables

Además de las asignaciones de valores a variables vista con anterioridad, que consistía en utilizar el operador de asignación (=), podemos asignarle valores dependiendo del estado de la variable.

### Uso de variable no definida o con valor nulo

Cuando una variable no está definida, o lo está pero contiene un valor nulo, se puede hacer que se use un valor predeterminado mediante la siguiente la expresión:

```
$ {variable:-valorpredeterminado}
```

Esta expresión devuelve el contenido de variable si está definida y tiene un valor no nulo. Por ejemplo si la variable *resultado* inicialmente no esta definida:

```
$ echo ${resultado}
$ echo "E1 resultado es: {resultado:-0}"
E1 resultado es: O
$ resultado=1
$ echo "E1 resultado es: ${resultado:-0}"
E1 resultado es: 1
```

A los parámetros posicionales podemos acceder como:

```
{$1: -o}
```

### Uso de variable no definida

En algunas ocasiones interesa utilizar un valor predeterminado sólo en el caso de que la variable no esté definida. La expresión que se utiliza para ello es algo diferente de la anterior:

```
${variable- valorpredeterminado}
```

Consultar el ejmplo anterior.

### Uso de variable definida o con valor nulo

Existe una expresión opuesta a la anterior. En este caso, si la variable está definida y contiene un valor no nulo, entonces en vez de usarse dicho valor, se utiliza el que se especifica. En caso contrario, el valor de la expresión es la cadena nula. La expresión para esto es:

```
$ {variable: +valorpredeterminado}
```

Por ejemplo:

```
$ resultado=10
$ echo ${resultado:+5}
5
$ resultado=
$ echo ${resultado:+30}

$
```

### Uso de variable no definida

En algunas ocasiones puede también interesar que el comportamiento anterior sólo suceda cuando la variable no esté definida. La expresión para ello es:

```
$ {variable+valorpredeterminado}
```

## Asignación de valores predeterminados de variables

Anteriormente veíamos la forma de utilizar un valor predeterminado de las variable en ciertos casos. Ahora, además de usar el valor predeterminado, queremos asignarlo a la variable.

### Asignación a variable o definida o con valor nulo

En este caso no sólo utilizamos un valor predeterminado, sino que en la misma operación lo asignamos. La expresión que se utiliza para ello es la siguiente:

```
${variable:=valorpredeterminado}
```

Si el contenido de *variable* es no nulo, esta expresión devuelve dicho valor. Si el valor es nulo o la variable no está definida entonces el valor de la expresión es *valorpredeterminado*, el cual será también asignado a la variable *variable*. Veamos un ejemplo en el que se supone que la variable *resultado* no está definida:

```
$ echo ${resultado}

$ echo "El resultado es: ${resultado:=0}"
E1 resultado es: 0
$ echo ${resultado}
0
```

A los parámetros posicionales no se le pueden asignar valores utilizando este mecanismo.

### Asignación a variable no definida

Análogo al caso anterior para el caso de que la variable no esté definida. La expresión ahora es:

```
${variable=valorpredeterminado}
```

## Mostrar un mensaje de error asociado a una variable

Ahora lo que pretendemos es terminar un script con un mensaje de error asociado al contenido de una variable.

### Variable no definida o con valor nulo

En otras ocasiones no interesa utilizar ningún valor por defecto, sino comprobar que la variable está definida y contiene un valor no nulo. En este último caso interesa avisar con un mensaje y que el programa de shell termine. La expresión para hacer esto es:

```
$ {variable : ?Mensaje }
Por ejemplo:
$ res=${resultado:? "variable no válida”}
resultado variable no valida
```

En el caso de que la variable resultado no esté definida o contenga un valor nulo, se mostrará el mensaje especificado en pantalla, y si esta instrucción se ejecuta desde un programa de shell, éste finalizará.

### Variable no definida

Análogo al caso anterior para el caso de que la variable no esté definida. La expresión para ello es:

```
${variable?mensaje}
```

## Otras operaciones con variables

Ciertas shell propporcionan unas facilidades que pueden ser útiles para ahorrar código en la programación de guiones de shell, como son la eliminación o extracción de subcadenas de una variable.

### Subcadenas de una variable

```
${variable:inicio:longitud}
```

Extrae una subcadena de *variable*, partiendo de *inicio* y de tamaño indicado por *longitud*. Si se omite *longitud* toma hasta el fin de la cadena original.

```
$ A=abcdef
$ echo ${A:3:2}
de
$ echo ${A:1:4}
bcde
$ echo ${A:2}
cdef
```

### Cortar texto al principio de una variable

```
${variable#texto}
```

Corta texto de variable si variable comienza por texto. Si variable no comienza por texto variable se usa inalterada. El siguiente ejemplo muestra el mecanismo de funcionamiento:

```
$ A=abcdef
$ echo ${A#ab}
cdef
$ echo ${A#$B}
cdef
$ B=abc
$ echo ${A#$B}
def
$ echo ${A#cd}
abcdef
```

### Cortar texto al final de una variable

```
${variable%texto}
```

Corta texto de variable si variable termina por texto. Si variable no termina por texto variable se usa inalterada.

Vemos un ejemplo:

```
$ PS1=$
$PS1="$ "
$ A=abcdef
$ echo ${A%def}
abc
$ B=cdef
$ echo ${A%$B}
ab
```

### Reemplazar texto en una variable

```
${variable/texto1/texto2}
${variable//texto1/texto2}
```

Sustituye texto1 por texto2 en variable. En la primera forma, sólo se reemplaza la primera aparición. La segunda forma hace que se sustituyan todas las apariciones de texto1 por texto2.

```
$ A=abcdef
$ echo ${A/abc/x}
xdef
$ echo ${A/de/x}
abcxf
```

## Evaluación aritmética

En habitual tener que efectuar evaluaciones de expresiones aritméticas enteras durante la ejecución de un script de shell; por ejemplo para tener contadores o acumuladores o en otros casos.

Hasta ahora habíamos visto que esto lo podíamos hacer con *expr*, pero hay otra forma más cómoda: *let*

La sintaxis de *let* es la siguiente:

```
let variable=expresión aritmética
```

por ejemplo

```
let A=A+1
```

En algunas shell incluso podremos omitir la palabra *let*, aunque por motivos de compatibilidad esto no es aconsejable.

Para evaluar expresiones reales, es decir con coma decimal, tendremos que usar otros mecanismos y utilidades que pueda proporcionar el sistema. En linux disponemos de la orden bc.

## Selección de la shell de ejecución

Si queremos que nuestro programa sea interpretado por una shell concreta lo podemos indicar en la primera línea del fichero de la siguiente forma

```
#!/ruta/shell
```

Por ejemplo, si queremos que sea la shell bash la que interprete nuestro script tendríamos que comenzarlo por

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

En ciertas ocasiones es interesante forzar que sea una shell concreta la que interprete el script, por ejemplo si el script es simple podemos seleccoinar una shell que ocupe pocos recursos como sh. Si por el contrario el script hace uso de características avanzadas puede que nos interese seleccionar bash como shell.

## Lectura desde la entrada estándar: read

La orden *read* permite leer valores desde la entrada estándar y asignarlos a variables de shell.

La forma más simple de leer una variable es la siguiente:

```
read variable
```

Después de esto, el programa quedará esperando hasta que se le proporcione una cadena de caracteres terminada por un salto de línea. El valor que se le asigna a variable es esta cadena (sin el salto de línea final).

La instrucción read también puede leer simultáneamente varias variables:

```
read varl var2 var3 var4
```

La cadena suministrada por el usuario empieza por el primer carácter tecleado hasta el salto de línea final. Esta línea se supone dividida en campos por el separador de campos definido por la variable de shell *IFS* (Internal Field Separator) que de forma predeterminada es una secuencia de espacios y tabuladores.

El primer campo tecleado por el usuario será asignado a var1, el segundo a var2, etc. Si el número de campos es mayor que el de variables, entonces la última variable contiene los campos que sobran. Si por el contrario el número de campos es mayor que el de variables las variables que sobran tendrán un valor nulo.

La segunda característica es la posibilidad incluir un mensaje informativo previo a la lectura. Si en la primera variable de esta instrucción aparece un carácter "?". todo lo que quede hasta el final de este primer argumento de read, se considerará el mensaje que se quiere enviar a la salida estándar.

Ejemplo:

```
read nombre?"Nombre y dos apellidos? " apl ap2
```

## Evaluación de expresiones: test

La orden test evalúa una expresión para obtener una condición lógica que posteriormente se utilizará para determinar el flujo de ejecución del programa de shell con las sentencias correspondientes. La sintaxis de la instrucción es:

```
test expr
```

Para construir la expresión disponemos una serie de facilidades proporcionadas por la shell. Estas expresiones evalúan una determinada condición y devuelven una

condición que puede ser verdadera o falsa. El valor de la condición actualiza la variable $? con los valores cero o uno para los resultados verdadero o falso.

Pasamos a describir a continuación estas condiciones, y tenemos que tener en cuenta que es importante respetar todos los espacios que aquí aparecen.

```
-f fichero existe el fichero y es un fichero regular
-r fichero existe el fichero y es de lectura
-w fichero existe el fichero y es de escritura
-x fichero existe el fichero y es ejecutable
-h fichero existe el fichero y es un enlace simbólico.
-d fichero existe el fichero y es un directorio
-p fichero existe el fichero y es una tubería con nombre
-c fichero existe el fichero y es un dispositivo de carácter
-b fichero existe el fichero y es un dispositivo de bloques
-u fichero existe el fichero y está puesto el bit SUID
-g fichero existe el fichero y está puesto el bit SGID
-s fichero existe el fichero y su longitud es mayor que O
-z s1 la longitud de la cadena s1 es cero
-n s1 la longitud de la cadena s1 es distinta de cero
s1=s2 la cadena s1 y la s2 son iguales
s1!=s2 la cadena s1 y la s2 son distintas
n1 -eq n2 los enteros n1 y n2 son iguales.
n1 -ne n2 los enteros n1 y n2 no son iguales.
n1 -gt n2 n1 es estrictamente mayor que n2.
n1 -ge n2 n1 es mayor o igual que n2.
nl -lt n2 n1 es menor estricto que n2.
nl -le n2 n1 es menor o igual que n2.
```

Estas expresiones las podemos combinar con:

*!* operador unario de negación

*-a* operador AND binario

*-o* operador OR binario

Ejemplos:

```
$ test -f /etc/profile
$ echo $?
$ test -f /etc/profile -a -w /etc/profile
$ echo $?
$ test 0 -lt 0 -o -n "No nula”
$ echo $?
```

La orden *test* se puede sustituir por unos corchetes abiertos y cerrados. Es necesario dejar espacios antes y después de los corchetes; este suele ser unos de los errores más frecuentes.

## Estructura de control

La programación en shell dispone de las sentencias de control del flujo de instrucciones necesarias para poder controlar perfectamente la ejecución de las órdenes necesrias.

### Sentencia if

La shell dispone de la sentencia *if* de bifurcación del flujo de ejecución de un programa similar a cualquier otro lenguaje de programación. La forma más simple de esta sentencia es:

```
if   lista_órdenes
then
   lista_órdenes
fi
```

*fi,* que es *if* alrevés, indica donde termina el *if*.

En la parte reservada para la condición de la sentencia *if*, aparece una lista de órdenes separados por ";". Cada uno de estos mandatos es ejecutado en el orden en el que aparecen. La condición para evaluar por if tomará el valor de salida del último mandato ejecutado. Si la última orden ha terminado correctamente, sin condición de error, se ejecutará la lista de órdenes que hay tras *then*. Si esta órden ha fallado debido a un error, la ejecución continúa tras el *if*.

Como condición se puede poner cualquier mandato que interese, pero lo más habitual es utilizar diferentes formas de la orden test. Por ejemplo:

```
if [ -f mifichero ]
then
echo "mifichero existe"
fi
```

Pero también podemos poner:

```
if grep body index.html
then
echo "he encontrado la cadena body en index.html"
fi
```

Como en cualquier lenguaje de programación también podemos definir las acciones que se tieenen que ejecutar en el caso de que la condición resulte falsa:

```
if lista_órdenes
then
        lista_órdenes
else
        lista_órdenes
fi
```

Por ejemplo:

```
if [ -f "$1" ] then
pr $1
else
echo "$1 no es un fichero regular"
fi
```

Cuando queremos comprobar una condición cuando entramos en el *else*, es decir, si tenemos else *if* es posible utilizar *elif*. Vemos un ejemplo:

```
if [ -f "$1" ]
then
cat $1
elif [ -d "$1" ]
then
ls $1/*
else
echo "$1 no es ni fichero ni directorio"
fi
```

### Sentencia while

La sentencia while tiene la siguiente sintaxis:

```
while lista_órdenes
do
lista órdenes
done
```

La lista de órdenes que se especifican en el interior del bucle *while* se ejecutará mientras que lista_órdenes devuelva un valor verdadero, lo que significa que la última orden de esta lista termina correctamente.

Vemos un ejemplo:

```
I=0
while [ ${resp:=s} = s ]
do
I=\‘{ }expr $I + 1\‘{ }
echo $I
read resp?"Quiere usted continuar(s/n)? "
done
```

### Sentencia until

La sentencia until similar a while, es otro bucle que se ejecutará hasta que se cumpla la condición, es decir, hasta que la lista de órdenes termina correctamente. Su formato es el siguiente:

```
until lista_órdenes
do
lista órdenes
done
```

### Sentencia for

La sentencia for repite una serie de órdenes a la vez que una variable de control va tomando los sucesivos valores indicado por una lista de cadenas de texto. Para cada iteración la variable de control toma el valor de uno de los elementos de la lista. La sintaxis de for es la siguientes

```
for variabl ein lista
do
lista mandatos
done
```

lista es una serie de cadenas de texto separadas por espacios y tabuladores. En cada iteración del bucle la variable de control variable toma el valor del siguiente campo y se ejecuta la secuencia de mandatos lista_mandatos.

Ejemplo:

```
for i in $*
do
echo $i
done
```

y mostraríamos todos los parámetros posicionales.

```
for i in *
do
echo $i
done
```

y mostraríamos la lista de ficheros del directorio activo.

### Sentencias break y continue

La sentencia break se utiliza para terminar la ejecución de un bucle *while* o *for*. Es decir el control continuará por la siguiente instrucción del bucle. Si existen varios bucles anidados, *break* terminará la ejecución del último que se ha abierto.

Es posible salir de n niveles de bucle mediante la instrucción *break n*.

La instrucción *continue* reinicia el bucle con la próxima iteración dejando de ejecutar cualquier orden posterior en la iteración en curso.

### Sentencia case

La sentencia case proporciona un *if* múltiple similar a la sentencia *switch* de C. El formato básico de esta sentencia es el siguiente:

```
case variable in
patrón1)
 lista_órdenes1
;;
patrón2)
lista_órdenes2
;;
...
patrónN)
lista_órdenesN;;
esac
```

La shell comprueba si variable coincide con alguno de los patrones especificados. La comprobación se realiza en orden, es decir empezando por *patrón1* terminando

por *patrónN*. En el momento en que se detecte que la cadena cumple algún patrón, se ejecutará la secuencia de mandatos correspondiente hasta llegar a ";;". Estos dos puntos y comas fuerzan a salir de la sentencia *case* y a continuar por la siguiente sentencia después de esac (esac es case alrevés).

Las reglas para componer patrones son las mismas que para formar nombres de ficheros, así por ejemplo, el carácter "*" es cumplido por cualquier cadena, por lo que suele colocarse este patrón en el último lugar, actuando como acción predeterminada para el caso de que no se cumpla ninguna de las anteriores. Ejemplo:

```
case "$1" in
  start)
        echo -n "Ha seleccionado start "
        ;;
  stop)
        echo -n "Ha seleccionado stop "
        ;;
  status)
        echo -n "Ha seleccionado stop "
        ;;
  restart)
        echo -n " Ha seleccionado restart "
        ;;
  *)
        echo "No es una opción válida"
        exit 1
esac
```

## Terminar un programa de shell (exit)

Como hemos visto, todas las órdenes órdenes tienen un estado de finalización, que por convenio es 0 cuando la ejecución terminó correctamente, y un valor distinto de 0 cuando lo incorrectamente o con error.

Si un programa de shell termina sin errores devolverá un valor cero, pero también es posible devolver explícitamente un valor mediante la sentencia exit. La ejecución de esta sentencia finaliza en ese instante el programa de shell, devolviendo el valor que se le pose como argumento, o el estado de la última orden ejecutada si no se le pasa ningún valor.

Ejemplo:

```
if grep "$1" /var/log/messages
then
exit 0
else
exit 10
fi
```

## Opciones en un programa de shell: getopts

En los sistema Unix es habitual poner las opciones para ejecutar una orden siguiendo unas normas:

- Las opciones están formadas por una letra precedida de un guión.

- El orden de las opciones es indiferente, pero suelen preceder a los ntes de los argumentos propiamente.
- Cada opción puede llevar uno o más argumentos.
- Las opciones sin argumentos pueden agruparse después de un único guión

Para facilitar el análisis de las opciones la shell dispone de la orden getopts. La sintaxis de getopts es la siguiente:

getopts cadenaopciones variable [args ...]

En *cadenaopciones* se sitúan las opciones válidas del programa. Cada letra en esta cadena significa una opción válida. El carácter ":" después de una letra indica que esa opción lleva un argumento asociado o grupo de argumentos separados por una secuencia de espacios y tabuladores.

Esta orden se combina con la sentencia *while* para iterar por cada opción que aparezca en la línea de órdenes en la llamada al programa. En *variable* se almacena el valor de cada una de las opciones en las sucesivas llamadas. En la variable OPTIND se guarda el índice del siguiente argumento que se va a procesar. En cada llamada a un programa de shell esta variable se inicializa a 1.

Además:

- Cuando a una opción le acompaña un argumento, getopts lo sitúa en la variable OPTARG.
- Si en la línea de mandatos apar ece una opción no válida entonces asignará "?" a variable.

Veamos un ejemplo:

```
while getopts ab:cd opcion
do
        case $opcion in
                a) echo "Opción a"
                ;;
                b) echo "Opción b con argumento OPTARG"
                ;;
                c) echo "Opción c"
                ;;
                d) echo "Opcion d"
                ?) echo "Uso: $0 -acd -b opcion "
        esac
done
shift ‘expr $OPTIND - 1‘
echo "resto de argumentos: $*"
```

Observamos como hemos desplazados todos los argumentos con shift para enerlos disponibles con los parámetros posicionales y descartando las opciones previamente analizadas.

## Evaluación de variables: eval

La orden eval toma una serie de cadenas como argumentos:

```
eval [arg1 [arg2] ...]
```

los expande siguiendo las normas de expansión de la shell, separándolos por espacios y trata de ejecutar la cadena resultante como si fuera cualquier orden.

Esta instrucción se debería utilizar cuando:

Pretendemos examinar el resultado de una expansión realizada por la shell.

Para encontrar el valor de una variable cuyo nombre es el valor de otra variable.

Ejecutar una línea que se ha leído o compuesto internamente en el programa de shell.

## Funciones

Ciertas shell como bash permiten la declaración de funciones para agrupar bloques código como en un lenguaje de programación convencional.

La forma de declarar un función es

```
function mi_funcion
{
código de la función
}.
```

Para realizar la llamada a la función sólo tenemos que usar su nombre. Además las funciones pueden tener argumentos en su llamada. No es necesario declarar los parámetros en la declaración de la función, basta usar las variables $1, $2, etc. dentro de la definición de las instrucciones y serán los parametros en su orden correspondiente. Para llamar a una función con argumentos no se usan los habituales paréntesis.

Ejemplos:

```
      #!/bin/bash
  function terminar
  {
                exit 0
      }
  function saludo
  {
   echo ¡Hola Mundo!
}
saludo
terminar
```

Otro ejemplo:

```
            #!/bin/bash
 function terminar
  {
                     exit 0
  }
  function saludo
  {
    echo $1
   }
    saludo Hola
    saludo Mundo
    terminar
```

En este último ejemplo podemos observar el uso de una función con argumentos, tanto en la delaración como en la llamada.

## Trucos de programación en shell

Vemos a continuación una serie de ejemplo genéricos para resolver cuestiones que se presentan en la programación de shell con cierta frecuencia

### Script con número variable de argumentos:

Este programa de shell pretende ejecutar una serie de instrucciones para cada uno de los argumentos que se le pasen desde la línea de órdenes.

```
for i in $*
do
instrucciones
done
```

Por ejemplo, hacemos un script que mueva todos los ficheros pasado como argumento al directorio *./papelera*:

```
for i in $*
do
if [ -f $i ]
then
mv $i ./papelera
fi
done
```

### Script para aplicar una serie de órdenes a cada fichero de un directorio

Muy similar al ejemplo anterior, pero sustituimos $* por simplemente * que equivale a todos los ficheros del directorio activo.

```
for i in *
do
instrucciones
done
```

### Leer un fichero de texto línea a línea

Es muy habitual tener que procesar todas las línea de un fichero de texto para realizar diferentes operaciones. Vemos una primera forma:

```
while read LINEA
do
instucciones por línea
done <fichero
```

En este caso estamos redirigiendo la entrada estándar de la orden *read*, que es el teclado, por un fichero. Al igual que en el caso del teclado, la lectura se realizará hasta que se encuentre un salto de línea. Observamos como la redirección se realiza tras el final de la sentencia *while*.

Otra forma posible para hacer esto mismo sería:

```
cat fichero|while read LINEA
do
instrucciones por línea
done
```

Este método difiere ligeramente del anterior, ya que al utilizar una tubería creamos una nueva shell con lo cual puede ocurrir que no se conserven ciertos valores de las variables de shell.

Cambiar una secuencia de espacios por un separador de campos

La salida de ciertas órdenes y ciertos ficheros separan los datos por espacios en blanco. Por ejemplo las órdenes *ps* o *ls -la*, o *ifconfig*.

Si queremos utilizar parte de los datos de las salidas de estas órdenes tendremos que contar las columnas en las que aparece cada dato y cortarlos con *cut* usando la opción *-c*. Pero otra opción sería sustituir toda una serie de espacios en blanco por un separador, por ejemplo ":" o ";".

Por ejemplo vamos a ver como sustituir los espacios de la orden ifconfig por ";".

La salida normal sería:

$ /sbin/ifconfig

```
eth0      Link encap:Ethernet  HWaddr 00:90:F5:08:37:E4
          inet addr:192.168.1.5  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5103 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:10 Base address:0x3200

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1726 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1726 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Ahora usamos sed para sustituir cualquier secuencia de espacios en blanco ([ ][ ]*) por un separador ";":

```
$ /sbin/ifconfig |sed "s/[ ][ ]*/;/g"
eth0;Link;encap:Ethernet;HWaddr;00:90:F5:08:37:E4;
;inet;addr:192.168.1.5;Bcast:192.168.1.255;Mask:255.255.255.0
;UP;BROADCAST;MULTICAST;MTU:1500;Metric:1
;RX;packets:0;errors:0;dropped:0;overruns:0;frame:0
;TX;packets:5241;errors:0;dropped:0;overruns:0;carrier:0
;collisions:0;txqueuelen:100;
;Interrupt:10;Base;address:0x3200;

lo;Link;encap:Local;Loopback;
;inet;addr:127.0.0.1;Mask:255.0.0.0
;UP;LOOPBACK;RUNNING;MTU:16436;Metric:1
;RX;packets:1773;errors:0;dropped:0;overruns:0;frame:0
;TX;packets:1773;errors:0;dropped:0;overruns:0;carrier:0
;collisions:0;txqueuelen:0;
Ahora podríamos cortar de forma exacta el campo que nos interese, por ejemplo:
$ /sbin/ifconfig | sed "s/[ ][ ]*/:/g" | grep inet | cut -f4 -d:
192.168.1.5
127.0.0.1
```

Vemos paso a paso la anterior orden compuesta:

- Primero ejecutamos la orden *ifconfig*
- Sustituimos los espacios en blanco por ":"
- Buscamos la línea que contenga la palabra *inet*
- Cortamos el campo 4 usando ":" como separador.

## Prácticas

Ejercicios propuestos

¿Que salida ocasionaría cada una de las siguientes órdenes si la ejecutamos consecutivamente?

```
$ set a b c d e f g h i j k l m n
$ echo $10
$ echo $15
$ echo $*
$ echo $#
$ echo $?
$ echo ${11}
```

Explica que realizaría cada una de las siguientes órdenes ejecutadas en secuencia

```
$ A=\$B
$ B=ls
$ echo $A
$ eval $A
```

Mostrar el último parámetro posicional

Pista: $#

Asignar el úlimo parámetro posicional a la variable ULT

Realizar un programa que escriba los 20 primeros números enteros.

Realizar un programa que numere las líneas de un fichero

Realizar un programa que tomando como base el contenido de un directorio escriba cada elemento contenido indicando si es fichero o directorio.

Realizar un programa que muestre todos los ficheros ejecutables del directorio activo.

Modificar el programa anterior para que indique el tipo de cada elemento contenido en el directorio activo: fichero, directorio, ejecutable,...

### Ejercicios resueltos sobre ficheros y directorios

Guion de shell que genere un fichero llamado listaetc que contenga los ficheros con permiso de lectura que haya en el directorio /etc:

```
for F in /etc/*
do
if [ -f $F -a -r $F ]
then
echo $F >> listaetc
```

```
fi
done
```

Hacer un guion de shell que, partiendo del fichero generado en el ejercicio anterior, muestre todos los ficheros del directorio /etc que contengan la palagra "procmail":

```
while read LINEA
do
if grep procmail $L >/dev/null 2>&1
then
echo $L
fi
done <listaetc
```

Hacer un guion de shell que cuente cuantos ficheros y cuantos directorios hay en el directorio pasado como argumento:

```
DI=0
FI=0
for I in $1/*
do
if [ -f $I ]
then
let FI=FI+1
fi
if [ -d $I ]
then
let DI=DI+1
fi
done
```

Hacer un guion de shell que compruebe si existe el directorio pasado como argumento dentro del directorio activo. En caso de que exista, que diga si no está vacío.

```
if [ -d $1 ]
then
echo "$1 existe"
N=$(ls | wc -l)
if [ $N -gt 0 ]
then
echo "S1 no está vacio, contiene $N ficheros no ocultos"
fi
fi
```

Hacer un guion de shell que copie todos los ficheros del directorio actual en un directorio llamado csg. Si el directorio no existe el guion lo debe de crear.

```
if [ ! -d csg ]
then
mkdir csg
fi
cp * csg
```

Hacer un script que muestre el fichero del directorio activo con más líneas:

```
NLIN=0
for I in *
do
if [ -f $I ]
then
N=$(wc -l $I)
if [ $N -gt $NLIN ]
then
NOMBRE=$I
NLIN=$N
fi
fi
done
echo "$NOMBRE tiene $NLIN lineas"
```

## Claves

### Como...

- Ver la lista de ficheros de un directorio
- Crear un directorio
- Borrar un directorio
- Borrar un fichero
- Cambiar el directorio activo
- Motrar un fichero en pantalla
- Modificar lo permisos de un fichero
- Modificar los permisos de los ficheros de nueva creacion
- Modificar el propietario de uun fichero
- Crear un fichero vacio
- Modificar el grupo de un fichero
- Ver la ultimas lineas de un fichero
- Ver la primeras lineas de un fichero
- Ordenar un fichero
- Mostrar un fichero poco a poco. Y ambi´en para mostrar un fichero poco a poco
- Cambiar el nombre a un fichero
- Copiar un fichero
- Ver el directorio activo
- Enlazar dos ficheros
- Ver las diferencias entre dos ficheros
- No permitie que alguien pueda ver un fichero
- No permitie que alguien pueda entrar en un directorio
- Impedir o permitir el acceso a un directorio
- Impedir o permitir la modificaci´on de un fichero

- Comprimir ficheros y otro método para comprimir ficheros
- Comprimir directorios
- Cambiar la clave de acceso
- Ver la lista de procesos activos en el sistema
- Cortar informacion de una linea
- Sustituir una palabra en un fichero
- Buscar una palabra en un fichero
- Continuar la ejecucion de un proceso en segundo plano
- Mostrar un calendario
- Ver la fecha
- Imprimir un fichero
- Traer un proceso a primer plano