Capítulo 2

El sistema de archivos

2.1. Concepto de archivo y de sistema de archivos

Podemos definir de forma genérica el término archivo como un conjunto de datos con un nombre asociado. Los archivos suelen residir en dispositivos de almacenamiento secundario, tales como cintas, discos rígidos o disquetes. La razón de asignar un nombre a cada archivo es que de este modo tanto los usuarios como los programas pueden hacer referencia a los mismos de una forma lógica. Los procesos o programas en ejecución disponen de un conjunto de funciones proporcionadas por el sistema operativo para poder manipular esos archivos. Ese conjunto de funciones se conoce con el nombre de llamadas al sistema o *system calls*. El concepto de llamada al sistema es más amplio, pues engloba también funciones relacionadas con la manipulación de procesos y dispositivos. Un proceso o programa en ejecución puede escribir datos en un archivo mediante la llamada al sistema *write* y leerlos más tarde, o bien dejarlos allí para que otros procesos puedan leerlos mediante la llamada al sistema *read*. También los procesos tienen la posibilidad de crear archivos, añadir o eliminar información en ellos, desplazarse dentro para consultar la información deseada, etc., a partir del correspondiente conjunto de llamadas al sistema. En cierto modo, se puede entender un archivo como una extensión del conjunto de datos asociados a un proceso, pero el hecho de que estos datos continúen existiendo aunque el proceso haya terminado, los hace especialmente útiles para el almacenamiento de información a largo plazo. Hemos comentado el concepto de llamada al sistema como mero apunte informativo; el usuario final no tiene por qué ser consciente de la existencia de tales llamadas, ya que existen aplicaciones de más alto nivel que son las que las manipulan adecuadamente.

Algunos sistemas operativos imponen a todos sus archivos una estructura determinada bien definida. En Linux un archivo no es más que una secuencia de bytes (8 bits). Algunos programas esperan encontrar estructuras de diferentes niveles, pero el núcleo (*kernel*) no impone ninguna estructura sobre los archivos. Por ejemplo, los editores de texto esperan que la información guardada en el archivo se encuentre en formato ASCII, pero el núcleo no sabe nada de eso.

Un sistema de archivos debemos entenderlo como aquella parte del sistema responsable de la administración de los datos en dispositivos de almacenamiento secundario.

El sistema de archivos debe proporcionar los medios necesarios para un almacenamiento seguro y privado de la información y, a la vez, la posibilidad de compartir esa información en caso de que el usuario lo desee.

Entre las características más relevantes del sistema de archivos de Linux podemos citar las siguientes:

- Los usuarios tienen la posibilidad de crear, modificar y borrar archivos y directorios.
- Cada archivo tiene definidos tres tipos de acceso diferentes: acceso de lectura `[r]`, acceso de escritura `[w]` y acceso de ejecución `[x]`.
- A su vez, esos tres tipos de acceso pueden extenderse a la persona propietaria del archivo, al grupo al cual está adscrita dicha persona y al resto de los usuarios del sistema. Eso permite que los archivos puedan ser compartidos de forma controlada.
- Cada usuario puede estructurar sus archivos como desee, el núcleo de Linux no impone ninguna restricción.
- Linux proporciona la posibilidad de realizar copias de seguridad de todos y cada uno de los archivos para prevenir la pérdida de forma accidental o maliciosa de la información.
- Proporciona la posibilidad de cifrado y descifrado de información. Eso se puede hacer para que los datos sólo sean útiles (legibles) para las personas que conozcan la clave de descifrado.
- El usuario tiene una visión lógica de los datos, es el sistema el encargado de manipular correctamente los dispositivos y darle el soporte físico deseado a la información. El usuario no tiene que preocuparse por los dispositivos físicos, es el sistema el que se encarga de la forma en que se almacenan los datos en los dispositivos y de los medios físicos de transferencia de datos desde y hacia los mismos.

En Linux los archivos están organizados en lo que se conoce como directorios. Un directorio no es más que un archivo algo especial, el cual contiene información que permite localizar otros archivos. Los directorios pueden contener, a su vez, nuevos directorios, los cuales se denominan subdirectorios. A la estructura resultante de esta organización se la conoce con el nombre de estructura en árbol invertido. Un ejemplo típico de árbol de directorios Linux lo tenemos representado en la figura 2.1

El sistema de archivos de Linux tiene, para el usuario, una estructura en árbol invertido en el cual los archivos se agrupan en directorios. En él, todos los archivos y directorios dependen de un único directorio denominado directorio raíz o *root*, el cual se representa por el símbolo *slash* "`/`". En caso de que tengamos varios dispositivos físicos de almacenamiento secundario en el sistema (normalmente discos o particiones de disco), todos deben depender del directorio raíz, como un subdirectorio que depende, directa o indirectamente, de la raíz. A esta operación se la conoce con el nombre de montaje de un subsistema de archivos.

Los archivos se identifican en la estructura de directorios por lo que se conoce como *pathname* o camino. Así, la cadena `/etc/passwd` identifica a `passwd` como un elemento que cuelga del directorio `etc` el cual a su vez cuelga del directorio raíz (`/`). A partir de la

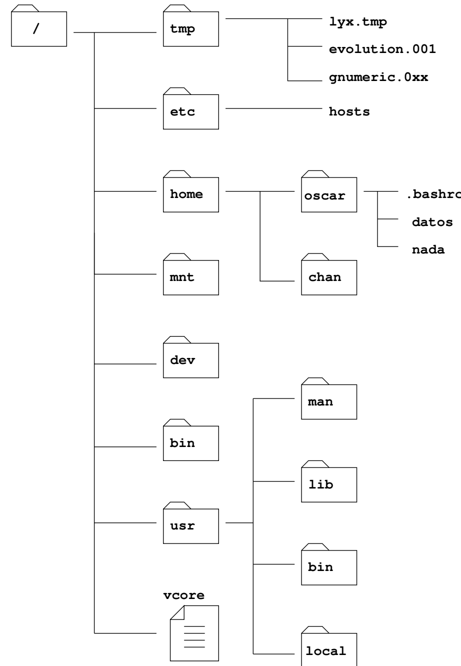


Figura 2.1: Esquema del árbol típico de directorios de Linux.

cadena `/etc/passwd` no podremos saber si `passwd` es un archivo o un directorio. Cuando el nombre del camino empieza con el carácter `/` se dice que el camino es absoluto. Linux también dispone de nombres de camino relativos, por ejemplo, si nuestro directorio actual es `/usr`, la cadena `bin/troff` identifica al archivo o directorio `/usr/bin/troff`. A esta cadena se la conoce, como hemos señalado antes, como camino relativo puesto que no comienza con el símbolo *slash*.

Cuando creamos un directorio, se crean automáticamente dos subdirectorios cuyos nombres son "`.`" (punto) y "`..`" (punto punto). "`.`" es una entrada en el directorio que identifica al directorio mismo y "`..`" es una entrada al directorio padre, es decir, aquel directorio del cual cuelga el subdirectorio actual. Las cadenas "`.`" y "`..`" también pueden ser utilizadas en el nombre de un camino relativo. Si por ejemplo actualmente estamos colocados en `/usr/lib`, la cadena `../include` identifica perfectamente al archivo o directorio `/usr/include`.

Ejemplos:

Si consideramos el archivo `xterm`, éste puede ser referenciado tanto por su ruta absoluta como por la relativa. La ruta absoluta es algo que no depende de nuestra posición actual, y es de la forma:

```
/usr/bin/X11/xterm
```

La ruta relativa depende del directorio en que nos encontremos en cada instante. Por ejemplo, si estuviésemos colocados en el directorio `/usr/lib`, la ruta relativa de `xterm` sería:

```
../bin/X11/xterm
```

Si estuviésemos en el directorio `/usr/bin`, la ruta relativa sería:

```
X11/xterm
```

Volvemos a insistir en este punto en el hecho de que Linux diferencia entre letras mayúsculas y minúsculas también para las rutas de archivos. Así, el directorio cuya ruta es `/usr/bin/X11` no es el mismo que `/usr/bin/x11`, en el caso de que este último existiese.

2.2. Algunos directorios interesantes

Todos los sistemas Linux, a diferencia de otros sistemas operativos, tienen una estructura de directorios estándar semejante a la representada en la figura 2.1. Seguidamente vamos a comentar algunos directorios que merecen mención especial.

`El directorio raíz /` Como hemos señalado antes, hay una, y sólo una, raíz en un sistema de archivos Linux y se denota por el carácter "/". La raíz es el único directorio que no tiene directorio padre. En este directorio las entradas "`.`" y "`..`" coinciden.

`/boot` En este directorio almacena un archivo que contiene la imagen binaria de arranque del núcleo de Linux; dicho de otro modo, contiene el código del propio sistema operativo Linux. Esta imagen se carga en memoria nada más iniciarlo, y se mantiene allí hasta que se apaga. El nombre de este archivo depende del sistema, pero unos nombres muy extendidos son `vmunix`, `Image`, `zImage` o `vmlinuz`. Es muy importante que no borremos este archivo, puesto que si lo hacemos, el sistema no podrá iniciarse. Solamente el administrador del sistema debe tener derecho para eliminar el archivo anterior.

`/bin` El directorio `/bin` (por binario) contiene muchas de las órdenes ejecutables utilizadas en Linux. Normalmente, aquí se encuentran los programas de uso más común para los usuarios, como la orden `/bin/cp` para copiar archivos, la orden `/bin/cat` para visualizar archivos de texto o la orden `/bin/ls` para visualizar los archivos de un determinado directorio.

`/home` Del directorio `/home` cuelgan los diferentes directorios de trabajo de cada uno de los usuarios. Cada usuario va a poder hacer lo que quiera con su directorio de trabajo (crear archivos, borrarlos, crear directorios, etc.), pero va a tener un acceso restringido al resto de los directorios. Un usuario normal, por ejemplo, no va a poder borrar un archivo del directorio raíz o copiar un programa en el directorio `/bin`.

`/usr` De este directorio colgaban en las primeras versiones de UNIX los subdirectorios de trabajo de los usuarios. Actualmente el directorio `/usr` contiene también archivos que posteriormente utilizan otras órdenes de Linux. De `/usr` cuelgan, además, algunos subdirectorios importantes como pueden ser:

`/usr/bin` Contiene fundamentalmente los programas ejecutables que de alguna forma son mayores en tamaño y se utilizan menos frecuentemente que las órdenes del directorio `/bin`.

`/usr/lib` Contiene los archivos de biblioteca utilizados por los compiladores de lenguajes como FORTRAN, Pascal, C, etc. Estos archivos contienen básicamente funciones, en un formato específico, que pueden ser invocadas desde estos lenguajes.

`/usr/man` Este directorio contiene las páginas del manual en el disco del ordenador. La orden `man`, que vimos en el capítulo anterior, lo único que hace es buscar en este directorio la información solicitada por el usuario y formatearla para que aparezca adecuadamente presentada por pantalla.

`/usr/local/bin` **y** `/usr/contrib/bin` Estos directorios son generalmente creados por el administrador del sistema para que contengan archivos ejecutables que no forman parte del Linux. Cualquier usuario que desarrolle una nueva utilidad, puede dejarla en uno de los dos directorios anteriores de modo que sea accesible al resto de los usuarios.

`/etc` Este directorio contiene órdenes y archivos de configuración empleados en la administración del sistema. Estas órdenes se guardan en un directorio aparte porque la mayoría de ellas sólo pueden ser ejecutadas por usuarios privilegiados. Normalmente, todos los archivos de configuración presentes en Linux son archivos de texto. La razón es que de este modo son fáciles de interpretar y de modificar, para lo cual necesitaremos únicamente un editor de texto.

`/dev` Este directorio contiene los archivos de dispositivo empleados para la comunicación con dispositivos periféricos, tales como cintas, impresoras, discos, terminales, etc. Un archivo de dispositivo es un archivo especial, reconocido por el núcleo, que representa a un elemento de entrada-salida (E/S). La idea de tratar los dispositivos de E/S como si se tratase de archivos es algo que se conoce con el nombre de independencia de dispositivo. La independencia de dispositivo es algo realmente interesante y, por otra parte, muy utilizado, porque de este modo emplearemos las mismas funciones tanto para trabajar con archivos ordinarios como para trabajar con elementos de E/S.

2.3. Nombres de archivos y directorios

Aunque ya hemos tratado con distintos nombre de archivos y directorios, todavía no sabemos qué reglas se utilizan para nombrarlos.

Los nombres de los archivos pueden contener hasta 255 caracteres. Los caracteres empleados pueden ser cualesquiera. En la práctica, sin embargo, se suelen evitar aquellos caracteres del código ASCII que tienen significado especial para el intérprete de órdenes. Como carácteres especiales podemos citar los siguientes:

```
* ? > < | [ ]  \ $ " ( ) etc.
```

Todos los nombres de archivos que figuran a continuación son nombres adecuados:

```
direcciones
listado_de_notas
carta_a_los_reyes_magos
ordenar.c
.profile
```

Si queremos evitar problemas de interpretación por parte del shell, no deberemos utilizar nombres de archivos como los que se indican seguidamente:

```
$dinero$
?datos
<desastre>
50|60_nombres
```

2.3.1. Convenios en los nombres de los archivos

A pesar de que el nombre de un archivo puede elegirse libremente, ciertas aplicaciones toman como convenio que los archivos con los cuales trabajan se diferencien del resto en algún rasgo identificador. Entre estas aplicaciones podemos citar los programas fuente escritos en un lenguaje de alto nivel. De este modo, un archivo que termine en `.c`, indica que contiene código fuente en lenguaje C. Si termina en `.f`, indica que contiene código fuente FORTRAN; si acaba en `.p`, se trata de un programa escrito en Pascal, etc. Esto no impide que alguien llame a un juego, por ejemplo, `juego.p`, aunque no se corresponda con un programa fuente escrito en Pascal.

Los convenios anteriores no afectan a los programas que contienen código ejecutable. Tales programas pueden tener cualquier nombre, lo que despista mucho a las personas que están acostumbradas a trabajar con sistemas operativos en los que los archivos ejecutables tienen algún rasgo diferenciador del resto de los archivos.

Obsérvese que al hablar del nombre de los archivos no hemos mencionado el concepto de extensión, empleado en otros sistemas. En Linux un archivo puede no tener extensión, tener una, dos o siete. Así pues, los siguientes nombres de archivo son perfectamente válidos en Linux:

```
programa.ejecutable.uno
prog.ver.1.1.0.3
```

2.4. Manipulación de archivos y directorios

Vamos a ver seguidamente una serie de órdenes empleadas para manipular archivos y directorios. Mostraremos cómo podemos movernos por los diferentes directorios, cómo ver el contenido de cada directorio, contenido, proteger la información, etc. La mayoría de las órdenes que vamos a ver en el resto del capítulo son de uso muy frecuente, y es bueno familiarizarse con ellas.

ls

```
Sintaxis: ls [-lFaRd] [archivo(s)]
```

La orden `ls` se utiliza para listar los archivos contenidos en un determinado directorio. Si no se le especifica ningún archivo ni directorio como argumento en la línea de órdenes, por defecto se visualizará el contenido del directorio de trabajo actual. Además, `ls` admite diversas opciones, las cuales son optativas, y permiten mostrar diversa información relacionada con los archivos. Sólo consideraremos las opciones más comunes, pero ni qué decir tiene que existen muchas otras. Si quisiéramos obtener toda la información acerca de la orden, tendríamos que servirnos del manual.

Ejemplo:

```
$ ls
Desktop    X         cfg     gzs      mail    rpm     va
KMail      a.out     doc     html     mbox    sigops  vst
Linux      acm       draw    http     mso     sisfi   xntp
LinuxDoc   autosave  errors  imlib    nsmail  tgz     xpdf
Mail       backup    exa     kdeinit  prac    tk
Tesis      c         fs      l4       ps      tmp
$
```

En algunos casos necesitaremos información adicional acerca de todo lo visualizado. En el ejemplo anterior no sabremos si el archivo `xpdf`, por ejemplo, es un archivo ordinario, un directorio o un programa ejecutable. Los archivos ejecutables en Linux no tienen ninguna extensión que los identifique, tal y como ocurre en otros sistemas operativos. Con la opción `-F`, `ls` añade un *slash* carácter "/" a cada directorio y un asterisco "*" a cada archivo que sea ejecutable.

Ejemplo:

```
$ ls -F
Desktop/    a.out{*}    draw/     imlib/     prac/     tmp/
KMail/      acm/        errors    kdeinit{*}  ps/       va/
Linux/      autosave/   exa/      l4/        rpm/      vst/
LinuxDoc/   backup/     fs/       mail/      sigops/   xntp/
Mail/       c/          gzs/      mbox       sisfi/    xpdf/
Tesis/      cfg/        html/     mso/       tgz/
X/          doc/        http/     nsmail/    tk/
$
```

En el caso anterior, queda claro que `kdeinit` y `a.out` son archivos ejecutables y `Desktop`, `Kmail` o `Linux` son directorios.

Cuando queremos una información lo más extensa posible de cada archivo, utilizaremos la opción `-l` para que se visualicen los archivos en formato largo.

Ejemplo:

```
$ ls -l
total 50
drwx------   6  chan   igx   1024 may 15 17:17 Desktop
drwx------   2  chan   igx   1024 nov 25 13:24 KMail
drwxr-xr-x   2  chan   igx   1024 nov 17 16:22 Linux
```

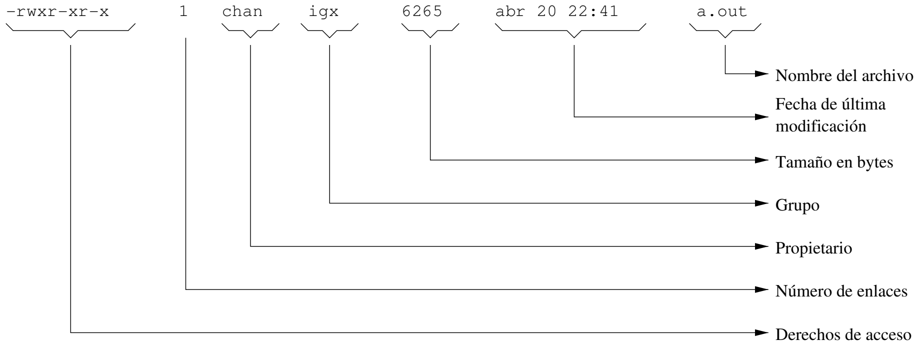


Figura 2.2: Información relacionada con un archivo Linux.

```
drwxr-xr-x   2  chan   igx   1024 nov 17 16:25 LinuxDoc
drwx------   2  chan   igx   1024 dec  4  2000 Mail
drwxr-xr-x   3  chan   igx   3072 nov  4 13:44 Tesis
drwxr-xr-x   2  chan   igx   1024 dec 13  2000 X
-rwxr-xr-x   1  chan   igx   5157 nov 17 16:31 a.out
drwxr-xr-x   5  chan   igx   1024 mar  7 21:48 acm
drwxrwxr-x   2  chan   igx   1024 abr 25 11:57 autosave
drwxr-xr-x   2  chan   igx   1024 abr  1 14:02 backup
drwxr-xr-x   2  chan   igx   1024 nov  3 10:03 c
drwxr-xr-x   2  chan   igx   1024 abr  2 17:23 cfg
drwxr-xr-x   2  chan   igx   1024 nov 28  2000 doc
drwxr-xr-x   2  chan   igx   1024 abr  1  2000 draw
drwxr-xr-x   3  chan   igx   1024 feb 26  1999 xntp
drwxr-xr-x   2  chan   igx   1024 may 29  2000 xpdf
$
```

La expresión `total 50` indica los bloques de disco (bloques de datos) ocupados por los archivos del directorio listado, que en este caso son 50. Generalmente el tamaño de bloque suele ser múltiplo de 512 bytes. Vamos a comentar a continuación cada uno de los campos que aparecen por cada archivo cuando damos la orden `ls` con la opción `-l`.

Los campos que aparecen por cada archivo (ver figura 2.2), de izquierda a derecha, son los siguientes:

- La primera columna comenzando por la izquierda es lo que se denomina modo de protección del archivo o lista de control de acceso. El primer carácter puede ser una "`d`", que indica que la entrada es un directorio,"`-`", que indica que se trata de un archivo ordinario. Si el archivo visualizado es un archivo de dispositivo (Linux trata a los dispositivos de entrada salida como si fuesen archivos), este primer carácter podrá ser una "`c`" o una "`b`", las cuales identifican a los archivos de dispositivo modo carácter y modo bloque, respectivamente. Los dispositivos modo carácter son aquellos en los que la transferencia de datos se hace carácter a carácter, como por ejemplo los terminales y las impresoras. Los dispositivos modo bloque son aquellos

que utilizan memorias intermedias (*buffers*) para realizar estas transferencias, como ejemplo típico tenemos los discos. El resto de los caracteres del modo del archivo (`rwxr-x--x`) informan de los permisos que el propietario, el grupo de personas al cual pertenece el propietario y el resto de los usuarios tienen sobre dicho archivo (consulte la orden **`chmod`** descrita en este mismo capítulo para obtener mayor información).

- Número de enlaces (*links*): un enlace permite que un archivo pueda aparecer en lugares diferentes en la estructura de directorio sin necesidad de tener su copia física repetida en el disco, lo que permite un mejor aprovechamiento del espacio. Para archivos normales, este número de enlaces suele ser 1. Si es mayor que 1, el número de enlaces indicará cuántas copias idénticas del archivo existen en los distintos directorios del sistema. Cuando el archivo es un directorio, el número de enlaces (*links*) indica cuántos subdirectorios tiene ese directorio.
- Nombre del propietario: muestra quién es el dueño del archivo o directorio. En el caso del ejemplo, el propietario es "`chan`".
- Nombre del grupo: indica el nombre del grupo al cual está adscrito el propietario del archivo. En el ejemplo es "`igx`".
- Tamaño del archivo: indica el número de bytes que contiene el archivo. En caso de que se trate de un archivo de dispositivo, aquí aparecerán el número mayor (*major number*) y el número menor (*minor number*). Estos números se emplean para identificar el propio dispositivo, y serán vistos con mayor profundidad más adelante.
- Fecha y hora de la última modificación: señala cuándo fue modificado por última vez o cuándo fue creado.
- Nombre del archivo: es el nombre del archivo o directorio.

Al hacer un listado, podemos observar que hay dos archivos que no aparecen, el directorio actual "`.`" y el directorio padre "`..`". Además, tampoco aparecerá ningún archivo cuyo primer carácter sea un punto. Si queremos ver tales archivos, tendremos que pasarle a `ls` la opción `-a`, que generalmente se combinará con la opción `-l`.

Ejemplo:

```
$ ls -al
total 181
drwx------  54  chan   igx   4096 nov 17 16:59 .
drwxr-xr-x  25  root   root  1024 nov 16 12:19 ..
-rw-------   1  chan   igx    161 may  8  2000 .Xauthorit
-rw-r--r--   1  chan   igx   1902 nov 20 12:30 .Xdefaults
drwx------   3  chan   igx   1024 nov  4 15:48 .ddd
-rw-r--r--   1  chan   igx    118 nov 22  2000 .desktop
drwxr-xr-x  12  chan   igx   1024 may  8  2000 .dt
etc.
$
```

La opción `-d` se utiliza normalmente junto con la opción `-l`. Esta opción la utilizaremos cuando queramos ver información relacionada con un directorio (propietario,

derechos, fecha, etc.) y no con su contenido (archivos o subdirectorios que cuelgan del directorio cuya información deseamos conocer).

Ejemplo:

```
$ ls -ld /etc
drwxr-xr-x  50  root   root  4096 jun 13 13:29 /etc
$
```

En el caso anterior, si no hubiésemos colocado el modificador `-d`, se hubiesen visualizado todos los archivos contenidos en el directorio `/etc` y no el directorio en sí.

pwd

Sintaxis: pwd

Esta orden muestra nuestro directorio de trabajo actual, tal y como indican sus iniciales (*print working directory*), en forma de camino absoluto. Cuando nos movemos mucho por el árbol de directorios, esta orden es de suma utilidad.

Si queremos evitar la consulta de nuestro directorio actual de trabajo continuamente, podremos hacer que el *prompt* muestre el camino donde estamos ubicados haciendo lo siguiente:

```
$ PS1=`$PWD> '
/home/chan>
```

PS1, como veremos más adelante, es una variable del shell que representa al *prompt*. Esto anterior funcionará si nuestro intérprete de órdenes o shell es el *bash* (*Bourne another shell*) o el *Korn* shell (*ksh*). El resultado no será el esperado si cambiamos letras mayúsculas por minúsculas. Para saber qué intérprete de órdenes estamos empleando, tenemos que ejecutar la orden `ps`. Si el shell que empleamos es el *Korn* o el *bash*, aparecerá una información similar a la siguiente:

```
$ ps
PID TTY           TIME CMD
 289 ttys000    0:00.07 -bash
 357 ttys000    0:00.75 vim esquema.otl
 636 ttys001    0:00.06 -bash
 938 ttys001    0:05.80 vim 02_SistemaArchivos/lyx/SistemaArchivos.tex
$ ps
$
```

Ejemplo de uso de la orden `pwd`:

```
$ pwd
/home/chan
$
```

En el ejemplo anterior, como podemos observar, estamos situados dentro del directorio `/home`, en un subdirectorio denominado `chan`.

cd

Sintaxis: cd [directorio]

La orden `cd` (*change directory*) se emplea para poder movernos de unos directorios a otros. El camino que le pasamos como argumento a `cd`, tal y como se muestra en la sintaxis, puede ser un nombre de camino absoluto o relativo. Si a `cd` no le pasamos como argumento ningún camino, nos localizará en nuestro directorio de arranque también conocido como directorio HOME (HOME es otra variable del shell). Al directorio anterior se le conoce como directorio de arranque o directorio de inicio, porque cuando iniciamos una sesión, el sistema automáticamente nos sitúa en ese punto.

Ejemplos:

```
$ pwd
/home/chan/doc
$ cd ..
$ pwd
/home/chan
$ cd /etc
$ pwd
/etc
$
```

Inicialmente estamos situados en el directorio `/home/chan/doc`, ejecutando la orden `cd ..` nos vamos al directorio padre (recuerde que "`..`" representa al directorio padre), que en este caso es `/home/chan`. No olvide el espacio en blanco después de `cd`, si no lo colocamos `cd` no funcionará y se visualizará un mensaje de error.

mkdir y rmdir

```
Sintaxis: mkdir directorio(s)
          rmdir directorio(s)
```

El árbol de directorios de Linux no es estático, sino que los usuarios tienen la posibilidad de crear sus propios directorios para distribuir mejor su información en el sistema. Los nuevos directorios no pueden ser creados en cualquier nodo del árbol. La mayoría de las veces, cada usuario sólo podrá crear nuevos directorios a partir de su directorio de inicio o directorio `HOME`; de esta manera, cada persona organiza como desee su información sin perjudicar al resto. Para crear un nuevo directorio, emplearemos la orden `mkdir` (*make directory*).

Ejemplo:

```
$ pwd
/home/chan/tmp
```

```
$ ls -al
total 12
drwxr-xr-x   2  chan   igx   4096 jun 17 17:50 .
drwx------  93  chan   igx   8192 jun 17 17:50 ..
$
```

Esto es lo que tenemos actualmente en el directorio de trabajo, si queremos crear un nuevo directorio para poder guardar nuestros programas fuentes en C podríamos hacer lo siguiente:

```
$ mkdir src
$ ls -al
total 16
drwxr-xr-x   3  chan   igx   4096 jun 17 17:52 .
drwx------  93  chan   igx   8192 jun 17 17:50 ..
drwxr-xr-x   2  chan   igx   4096 jun 17 17:52 src
$
```

Como podemos observar, en este caso `mkdir` crea un directorio nuevo a partir del actual. Si por cualquier causa queremos deshacernos de un directorio, utilizaremos la orden `rmdir` (*remove directory*). Antes de eliminar un directorio debemos asegurarnos de que dicho directorio está vacío. Siguiendo con el caso anterior, vamos a eliminar el directorio recién creado:

```
$ rmdir src
$ ls -al
total 12
drwxr-xr-x   2  chan   igx   4096 jun 17 17:55 .
drwx------  93  chan   igx   8192 jun 17 17:50 ..
$
```

cat

`Sintaxis: cat [archivo(s)]`

La orden `cat` sirve para visualizar el contenido de archivos de texto (ASCII) por la pantalla. Si a `cat` no le pasamos como argumento ningún archivo de texto, entonces leerá caracteres de la entrada estándar (teclado) hasta que pulsemos Ctrl-d (ˆd). Una vez hecho esto, visualizará lo que acabamos de escribir. Podemos observar que `cat` es una orden que por defecto (si no le pasamos ningún argumento) lee en la entrada estándar y dirige su salida a la salida estándar (pantalla). Más tarde veremos que a toda orden que cumpla estos requisitos se la conoce con el nombre de filtro. El carácter Ctrl-d en Linux es la marca de final de archivo. En el caso anterior, al pulsar la combinación de teclas indicada, marcamos el final de la entrada de datos desde el teclado.

Ejemplo:

```
$ cat prog.c
#include <stdio.h>
main (int argc, char* argv[])
{
    int x;
    for (x = 0; x < argc; x++)
        puts(argv[x]);
}
$
```

2.4.1. ¿Cómo podemos controlar la salida del terminal?

Determinadas órdenes pueden provocar un volcado masivo de información a la pantalla (éste es el caso de `cat` cuando visualizamos un archivo grande). En estos casos, la información pasa tan rápido que no somos capaces de leer nada. Si queremos detener ese volcado de información, podremos hacerlo utilizando la combinación de teclas Ctrl-s. Para reanudar de nuevo la visualización, pulsaremos Ctrl-q. Si lo que deseamos es abortar la orden definitivamente, utilizaremos la combinación de teclas Ctrl-c.

more

Sintaxis: more [archivo(s)]

La orden `more` imprime por pantalla el contenido del archivo de texto que le pasemos como argumento. En este caso, y a diferencia de lo que ocurría con `cat`, que mostraba todo el archivo de forma continua, la visualización se hace pantalla a pantalla.

Cuando `more` detiene la visualización, para poder continuar con la pantalla siguiente debemos pulsar la barra espaciadora. Si lo único que queremos es ver la siguiente línea, pulsaremos ENTRAR, y si queremos terminar la visualización, pulsaremos la tecla `q` (*quit*). En todo momento `more` nos va informando sobre qué tanto por ciento del tamaño del archivo lleva mostrado.

Ejemplo:

```
$ more serv.c

/*******************************************************
* Antes de iniciar el servidor y los clientes hay que   *
* crear cuatro fifos de nombres: Fifo1, Fifo2, Fifo3    *
* y Fifo4, mediante la orden mknod "Fifo# pchar" en el  *
* mismo directorio donde están tanto los clientes       *
* como el servidor.                                     *
*******************************************************/

#include <stdio.h>
#include <fcntl.h>
```

```
main()
{
  int DescFifo1, DescFifo2, DescFifo3, DescFifo4;
  int CanalActivo, nwrite;
  char ch;

  /* Abrimos los cuatro fifos en modo sólo escritura */

  if ((DescFifo1 = open ("Fifo1", O\_WRONLY)) == -1)
  {
    perror ("Error de apertura del Fifo 1");
--More--(36%)
```

head y tail

```
Sintaxis: head [-N] archivo(s)
          tail [-N] archivo(s)
```

Las órdenes `head` y `tail` se pueden utilizar para visualizar las primeras `N` líneas o las últimas `N` líneas de un archivo de texto, respectivamente. Esto puede ser útil, porque muchas veces no necesitamos visualizar el archivo de texto por completo, sino que nos basta con algunas líneas.

Ejemplos:

```
$ head -5 prog.c

#include <stdio.h>
main (int argc, char *argv[])
{
    int x;
$
```

En el ejemplo anterior visualizamos las primeras cinco líneas del archivo de texto `prog.c`.

```
$ tail -4 prog.c
    for (x = 0; x < argc; x++)
        puts(argv[x]);
    }
$
```

En este caso hemos visualizado las cuatro últimas líneas del archivo `prog.c`.

od

```
Sintaxis: od [-bcdfox] [archivo(s)]
```

La orden `od` (volcado octal, *octal dump*) se utiliza para realizar un volcado, en octal, del contenido de un archivo. Si a `od` no se le especifica ningún archivo, leerá de la entrada estándar hasta detectar el final de archivo Ctrl-d, y después visualizará lo escrito, en octal. Con la orden `cat` sólo podemos visualizar archivos de texto. Con `od` podemos visualizar el contenido de cualquier archivo, incluidos, por supuesto, los archivos de texto.

La orden `od` acepta diversas opciones; las más comunes son las siguientes:

`-b` Visualiza los bytes como números en código octal.

`-c` Visualiza los bytes como caracteres.

`-d` Visualiza las palabras (16 bits) como números decimales sin signo.

`-f` Visualiza el contenido del archivo como números en coma flotante de 32 bits.

`-o` Visualiza las palabras como números en octal sin signo (opción por defecto).

`-x` Visualiza las palabras en código hexadecimal.

Ejemplos:

```
$ od -c datos
0000000   C   o   n   t   e   n   i   d   o       d   e   l       a   r
0000020   c   h   i   v   o       "   d   a   t   o   s   "  \n   C   i
0000040   f   r   a   s   :  \t   1   2   3   4   5   6   7   8   9   0
0000060  \n  \n
0000062
$
$ od -b datos
0000000   103 157 156 164 145 156 151 144 157 040 144 145 154 040 141 162
0000020   143 150 151 166 157 040 042 144 141 164 157 163 042 012 103 151
0000040   146 162 141 163 072 011 061 062 063 064 065 066 067 070 071 060
0000060   012 012
0000062
$
$ od -bc datos
0000000   103 157 156 164 145 156 151 144 157 040 144 145 154 040 141 162
            C   o   n   t   e   n   i   d   o       d   e   l       a   r
0000020   143 150 151 166 157 040 042 144 141 164 157 163 042 012 103 151
            c   h   i   v   o       "   d   a   t   o   s   "  \n   C   i
0000040   146 162 141 163 072 011 061 062 063 064 065 066 067 070 071 060
            f   r   a   s   :  \t   1   2   3   4   5   6   7   8   9   0
0000060   012 012
           \n  \n
0000062
$
```

En el primer caso, hemos hecho un volcado del archivo `datos`, en el cual se visualizan los bytes del mismo como caracteres ASCII. El carácter `\n` es el carácter de nueva línea, y el carácter `\t` es el tabulador. Como se puede apreciar, la primera columna indica el

desplazamiento dentro del archivo (en octal). En el segundo caso hemos hecho otro volcado, pero ahora la visualización de cada byte se hace en forma de código octal. Del modo anterior podremos saber la correspondencia entre carácter ASCII y código octal asociado. En el tercer ejemplo, hemos utilizado las dos opciones anteriores simultáneamente. Aquí se puede apreciar aún mejor la correspondencia entre carácter ASCII y código octal asociado. Por ejemplo, el carácter a es el 141 en octal, y el carácter blanco es el 40 en octal.

cp

Sintaxis: cp archivo(s) destino

La orden `cp` se utiliza para copiar archivos de un lugar a otro en el árbol de directorios. Como mínimo, `cp` necesita dos argumentos, el primero es el archivo existente que queremos copiar en otro lugar, y el segundo es el nombre del destino. Las rutas de los dos archivos se pueden dar tanto de forma absoluta como relativa. Debemos tener cuidado a la hora de elegir el nombre del archivo destino, pues si previamente existía otro archivo con el mismo nombre, el original será sobreescrito. Si el nombre del archivo destino es un directorio, hará que el archivo fuente se copie dentro de dicho directorio con el mismo nombre que tenía el archivo original. Con `cp` también podemos copiar varios archivos fuente simultáneamente en un determinado directorio, destino debe ser obligatoriamente un directorio.

Ejemplo:

```
$ pwd
/home/chan/tmp
$ ls
datos prog prog.c serv.c
$ cp serv.c /home/chan/src/otro.c
$ cd ../src
$ ls
otro.c
$
```

Con ello hemos conseguido copiar el archivo `/home/chan/tmp/serv.c` en el directorio `/home/chan/src`, pero en este caso con el nombre `otro.c`.

mv

Sintaxis: mv archivo(s) destino

Esta orden tiene una sintaxis idéntica a `cp`. Con `mv`, lo que hacemos es mover los archivos de un lugar a otro. Como consecuencia, los archivos origen desaparecerán de su localización inicial. La orden `mv` la utilizaremos también para cambiar el nombre (renombrar) a un archivo. Para renombrar un archivo, no tendremos más que moverlo dentro del directorio en que esté localizado y éste adquirirá el nombre del archivo destino pasado como argumento.

Ejemplo:

```
$ pwd
/home/chan/tmp
$ ls
datos prog prog.c serv.c
$ mv prog.c ../src
$ ls
datos prog serv.c
$ cd ../src
$ ls
otro.c prog.c
$
```

Al mover el archivo `prog.c` desde el directorio `/home/chan/tmp` hasta el nuevo directorio `/home/chan/src`, vemos cómo el archivo inicial desaparece del directorio de origen. Al analizar el contenido del directorio destino, comprobamos que existe un nuevo archivo, denominado `prog.c`.

ln

Sintaxis: ln archivo(s) destino

La orden `ln` (*link*) tiene una sintaxis similar a las dos anteriores. Se utiliza para permitir que un mismo archivo aparezca en el sistema de archivos bajo dos nombres diferentes, pero con una única copia. Con `ln` no se hace una copia del archivo origen, solamente se crea otro nombre de archivo que hace referencia al mismo archivo físico. Eso permite que una única copia de un archivo aparezca en varios directorios con distintos nombres. De este modo, se puede compartir información de forma cómoda. Si en un momento eliminamos alguno de los archivos que hacen referencia a la misma copia física, sólo eliminaremos el nombre, pero no la copia real. Ésta sólo será definitivamente suprimida si eliminamos todos sus vínculos (*links*). El número de enlaces de un archivo lo indica el segundo campo de la información que obtenemos con la orden `ls -l`.

Ejemplo:

```
$ pwd
/home/chan/tmp
$ ls -l
total 8
-rw-r--r--   1  chan   igx     39 nov 18 16:05 datos
-rwxr-xr-x   1  chan   igx   4098 nov 17 18:24 prog
-rw-r--r--   1  chan   igx   1941 nov 17 18:29 serv.c
$

$ ln prog programa
$ ls -l
total 13
-rw-r--r--   1  chan   igx     39 nov 18 16:05 datos
-rwxr-xr-x   2  chan   igx   4098 nov 17 18:24 prog
```

```
-rwxr-xr-x   2  chan   igx   4098 nov 17 18:24 programa
-rw-r--r--   1  chan   igx   1941 nov 17 18:29 serv.c
$
```

En el ejemplo podemos ver cómo el campo que hace referencia al número de vínculos o enlaces varía de uno a dos, del primer al segundo ejemplo en el archivo `prog`. A partir de este momento, `prog` y `programa` son dos archivos diferentes que contienen la misma información y una única copia en el disco.

Vamos a insistir un poco más en esta orden, con objeto de dejar más claro su funcionamiento. Supongamos que tenemos un archivo, que denominamos `pss`. Usando la orden `ls -i` podemos visualizar su número de nodo-i. El número de nodo-i es un valor interno utilizado por el sistema de archivos que permite localizar toda la información relacionada con el propio archivo (tamaño, propietario, grupo, derechos de acceso, tipo de archivo, punteros a los bloques de disco, etc.).

```
$ ls -i pss
147468 pss
$
```

Nuestro archivo `pss` tiene un número de nodo-i igual a 147468 en el sistema de archivos. Ahora vamos a crear otro enlace a `pss` denominado `masp`. Para ello, daremos la orden:

```
$ ln pss masp
$
```

Vamos a ver de nuevo el número de nodo-i para el archivo enlazado `masp`.

```
$ ls -i masp
147468 masp
$
```

Como podemos comprobar, ambos archivos tienen el mismo número de nodo-i, de manera que accediendo a `pss` o a `masp` estamos accediendo al mismo archivo físico, ya que el sistema de archivos utiliza el mismo identificador de nodo-i en ambos casos. Cualquier cambio realizado en el primero de ellos se manifestará en el segundo, y viceversa.

A este tipo de enlaces se los conoce con el nombre de enlaces fuertes o *hard links*. El problema de este tipo de enlaces es que no sirven para archivos que se encuentren en sistemas de archivos diferentes (por ejemplo, diferentes particiones del disco). Los enlaces duros tampoco no son aplicables a directorios. Para solventar estos problemas, podemos hacer uso de otro tipo de enlaces, denominados enlaces simbólicos o *soft links*. Un enlace simbólico tiene una funcionalidad similar a un enlace duro, pero es posible utilizarlo en archivos que se encuentren en diferentes sistemas de archivos así como enlazar directorios. Para crear enlaces simbólicos, se utiliza la orden `ln` con la opción `-s` (*soft*).

Ejemplo:

```
$ ln -s pss assp
$
```

De esta forma, hemos creado un enlace a `pss` apuntado por `assp`. Si ahora utilizamos la orden `ls -i`, comprobaremos que ambos archivos tienen un número de nodo-i diferente:

```
$ ls -i pss assp
147469 assp 147468 pss
$
```

Utilizando la orden `ls -l`, podremos comprobar cómo `masp` es un enlace al primer archivo:

```
$ ls -l pss assp
lrwxrwxrwx   1  chan   igx      3 nov 19 17:48 assp -> pss
-rw-r--r--   2  chan   igx   4098 nov 19 17:50 pss
$
```

La primera `l` incluida junto con el campo de derechos del archivo `assp` indica que este archivo es un enlace simbólico a `pss`. Los permisos de un enlace simbólico no se utilizan (aparecen siempre a `rwxrwxrwx`). En estos casos, los derechos del archivo enlace son los mismos que los del archivo destino (en nuestro caso `pss`). En este caso, también tanto `pss` como `assp` hacen referencia a la misma información. Debemos tener cuidado con los enlaces simbólicos, ya que si eliminamos el archivo que actúa como destino del enlace, el archivo que lo enlazaba seguirá existiendo y apuntará a un archivo no existente. Esto es así porque el sistema, al contrario de lo que ocurría con los enlaces duros, no mantiene constancia del número de veces que un archivo se encuentra enlazado simbólicamente en el sistema de archivos.

rm

Sintaxis: `rm [-irf] archivo(s)`

La orden `rm` (*remove*) se utiliza para borrar archivos. Si alguno de los archivos referenciados no existiera, `rm` nos enviará un mensaje de aviso. Si el archivo no tiene derecho de escritura, aunque seamos su propietario, `rm` nos preguntará si realmente queremos eliminarlo. De otro modo, esta orden llevará a cabo su labor silenciosamente, sin enviarnos ningún mensaje. Debemos tener mucho cuidado con lo que vamos a borrar, puesto que Linux no permite que un archivo borrado pueda ser recuperado.

Las opciones más comunes de `rm` son:

`-f` (*force*) Fuerza el borrado de los archivos, incluso si están protegidos contra escritura (el archivo debe pertenecer al usuario que quiere borrarlo).

`-i` (*interactive*) Antes de borrar cada uno de los archivos, `rm` nos pregunta si realmente queremos hacerlo.

`-r` (*recursive*) Con esta opción `rm` borra los archivos de un directorio de forma recursiva, es decir, borra todos los posibles archivos localizados en subdirectorios dependientes del directorio especificado.

Ejemplos:

```
$ ls
assp    datos    masp    prog    programa  pss    serv.c
$ rm programa
$ ls
assp    datos    masp    prog    pss       serv.c
$
```

file

Sintaxis: file archivo(s)

Como hemos indicado anteriormente, Linux no impone ningún formato especial a sus archivos. El formato depende únicamente de los programas o utilidades que utilizan dicho archivo. Como hemos visto antes, `cat`, `head` y `tail` trabajan con archivos de texto (en código ASCII), pero no pueden trabajar con archivos de otro tipo, ya que estas órdenes interpretan sólo archivos de texto. La orden `file` intenta darnos información acerca del tipo del archivo que le pasemos como argumento. Para determinar los tipos, `file` lee unos cuantos bytes al comienzo del archivo, y a partir de esto busca indicios que le indiquen el tipo de archivo. Los archivos ejecutables puros son fáciles de identificar, puesto que en su comienzo llevan una marca, denominada número mágico o *magic number*, que identifica al archivo como tal. Si el archivo contiene ciertos patrones, tales como la cadena `main()`, `file` identificará al archivo como un programa fuente en lenguaje C. Estos indicios, que algunas veces se encuentran más escondidos, son los que busca la orden `file` para identificar el tipo de un archivo.

Ejemplo:

```
$ file /etc/passwd assp prog.c
/etc/passwd: ASCII text
assp: symbolic link to pss
prog.c: ISO-8859 C program text
$
```

2.5. Uso de archivos: permisos

El sistema Linux proporciona la posibilidad de proteger la información. Para ello, asocia a cada archivo una serie de derechos de acceso. En función de éstos, se determina qué es lo que cada usuario puede hacer con el archivo. Estos derechos se extienden a tres grupos de individuos: el propietario, el grupo del propietario y el resto. A su vez, estos grupos pueden tener diferentes posibilidades de acceso al archivo: para leer información del mismo, para escribir en él o para ejecutarlo, en el caso de que se corresponda con un archivo ejecutable. Estos derechos aparecen como una secuencia de nueve caracteres `r`, `w`, `x` o `-`. Una `r` indica derecho de lectura, una `w` de escritura, y la `x` de ejecución. El guión indica que el derecho correspondiente está desactivado. Estas secuencias de caracteres se agrupan de tres en tres. De izquierda a derecha tenemos lo siguiente: los tres primeros caracteres se corresponden con los derechos del propietario (*user*), los tres siguientes con los del grupo (*group*) y los tres últimos para el resto (*others*).

chmod

`Sintaxis: chmod modo archivo(s)`

La orden `chmod` (*change mode*) va a permitirnos modificar los permisos de un archivo. Para poder modificar estos derechos, debemos ser los propietarios del mismo. También el administrador del sistema o superusuario tiene la posibilidad de cambiarlos. Si no somos ni el propietario del archivo ni el administrador, `chmod` fallará. Para cambiar el modo de un archivo seguiremos estos pasos:

1. Convertir los campos de protección a dígitos binarios, poniendo un 1 en el caso de que queramos activar dicho campo (`rwx`), o un 0 en el caso de querer desactivarlo. Si, por ejemplo, queremos que los permisos finales del archivo sean `rwxr-xr--`, la secuencia de dígitos binarios sería: `111101100`.

2. Dividir esos dígitos binarios en tres partes de tres bits cada una: una para el usuario (propietario), otra para el grupo y una última para el resto de los usuarios (otros), de tres dígitos cada uno.

3. Convertir cada grupo de tres dígitos a numeración octal.

4. Reunir los tres dígitos octal en un único número, el cual será el modo que le pasemos como argumento a `chmod`.

5. Si, por ejemplo, queremos dejar un archivo con el modo `rwxr-xr--`, lo haremos de la siguiente forma:

Modo	Usuario	Grupo	Otros
`rwxr-xr--`	`rwx`	`r-x`	`r--`
Valor binario	`111`	`101`	`100`
Valor octal	7	5	4

Ejemplo:

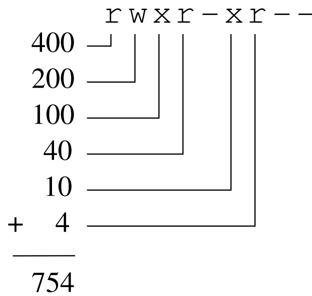
```
$ ls -l spcrun
-rw-r--r--   1  chan   igx   4098 nov 20 13:05 spcrun
$ chmod 754 spcrun
$ ls -l spcrun
-rwxr-xr--   1  chan   igx   4098 nov 20 13:05 spcrun
$
```

Otra forma de obtener el mismo resultado sería asignando a cada permiso de lectura, escritura y ejecución de cada usuario, grupo y otros, un número determinado y obtener el modo final que le pasamos como argumento a `chmod` sumando dichos números. Los valores asociados son los siguientes:

- Derecho de lectura del usuario, 400
- Derecho de escritura del usuario, 200

- Derecho de ejecución del usuario, 100
- Derecho de lectura del grupo, 40
- Derecho de escritura del grupo, 20
- Derecho de ejecución del grupo, 10
- Derecho de lectura del resto, 4
- Derecho de escritura del resto, 2
- Derecho de ejecución del resto, 1

Siguiendo con el ejemplo anterior, si queremos obtener la siguiente lista de permisos: `rwxr-xr--`, tendríamos que sumar:



Como podemos observar, el modo 754 es el mismo que el obtenido utilizando la técnica anterior.

Existe una última forma de especificar los modos de forma simbólica. Veamos unos ejemplos:

Ejemplos:

```
$ ls -l foo
-rwxr-xr--   1  chan   igx     54 nov 20 13:19 foo
$ chmod -w foo Quitamos el derecho de escritura al archivo foo
$ ls -l foo
-r-xr-xr--   1  chan   igx     54 nov 20 13:19 foo
$ chmod o+x foo Añadimos el derecho de ejecución para el resto de usuarios
$ ls -l foo
-r-xr-xr-x   1  chan   igx     54 nov 20 13:19 foo
$
```

De forma general, las abreviaturas simbólicas que podemos utilizar son las siguientes:

`u` Usuario

`g` Grupo

`o` Otros

`+` Añadir permiso

`-` Quitar permiso

umask

`Sintaxis: umask [máscara]`

Los permisos asignados a un archivo o a un directorio cuando son creados dependen de una variable denominada *user mask*. Podemos visualizar dicha variable dando la orden `umask` sin argumentos. El resultado son tres dígitos octales que indican, de izquierda a derecha, el valor de la máscara que determina los permisos iniciales para el propietario, para el grupo y para el resto de los usuarios. Cada dígito octal de la máscara contiene tres dígitos binarios, un 1 binario indica que cuando se cree un nuevo archivo el permiso asociado (`rwx`) será borrado, y un cero binario indica que se utiliza el permiso implícito. El permiso implícito es un permiso global que por defecto suele tener el valor `rw-rw-rw-` (modo 666). Si no deseamos que por defecto nuestros archivos y directorios se creen con estos valores, podremos cambiar el valor de la máscara de usuario dando la orden `umask` con el argumento oportuno. El valor del mismo puede ser calculado restando el modo deseado por defecto del modo actual. Por ejemplo, si queremos que nuestro modo por defecto para los nuevos archivos sea `rw-r-----` (640), entonces:

```
  666
- 640
 ----
  026
```

Donde:

`666` es el valor por defecto

`640` es el valor deseado

`026` es el argumento necesario para `umask`

Ejemplo:

```
$ umask 26
$
```

A partir de ahora todos los nuevos archivos que creemos tendrán los permisos siguientes: `rw-r-----`.

```
$ umask 26
$ umask
026
$ > prueba1
```

```
$ ls -l prueba1
-rw-r-----   1  chan   igx      0 nov 20 13:42 prueba1
$
$ umask 22
$ > prueba2
$ ls -l prueba2
-rw-r--r--   1  chan   igx      0 nov 20 13:43 prueba2
$
```

which

`Sintaxis: which archivo(s)`

Esta orden se emplea para buscar en los directorios especificados en el `PATH` de usuario el archivo que le especifiquemos. Como resultado, visualiza en forma de camino absoluto el nombre del archivo. Si la búsqueda es infructuosa, seremos avisados de ello.

Ejemplo:

```
$ which vi emacs pine
/bin/vi
/usr/bin/emacs
/usr/bin/pine
$
```

whereis

`Sintaxis: whereis [-b] [-m] [-s] orden(es)`

La orden `whereis` acepta como parámetro únicamente el nombre de una orden. Devuelve el directorio donde reside dicha orden y la página correspondiente donde se encuentra en el manual. Los flags `-b`, `-m` y `-s` se utilizan para limitar la búsqueda a binario, página del manual o código fuente, respectivamente.

Ejemplo:

```
$ whereis vi
vi: /bin/vi /usr/share/man/man1/vi.1.gz
$
```

id

`Sintaxis: id [-ug] [usuario]`

La orden `id` devuelve el identificador (número) de usuario y de grupo del usuario que le indiquemos. Si no se le indica el usuario, `id` visualizará los identificadores asociados al usuario que invoca la orden. Estos identificadores los utiliza Linux para saber a quién

tiene que aplicar los permisos. `id` es una orden intrínseca del shell. Que una orden sea intrínseca del shell quiere decir que se trata de una rutina incorporada dentro del código del propio intérprete de órdenes. No existe como programa ejecutable aparte, como puede ser `cp`, `man` o `mkdir`.

Opciones:

`-u` Visualiza sólo el UID (identificador de usuario).

`-g` Visualiza únicamente el GID (identificador de grupo).

Ejemplos:

```
$ id
uid=504(chan) gid=504(igx) grupos=504(igx)
$ id lucas
uid=519(lucas) gid=519(lucas)
$
```

Si el usuario indicado a `id` no existe, `id` visualizará un mensaje similar al siguiente:

```
$ id pascual
id:
$
```

su

Sintaxis: su [-] [usuario]

La orden `su` (*switch user*) permite cambiar nuestro identificador de usuario. Cuando se invoca, nos pide la palabra clave (*password*) del usuario al que queremos cambiar. Si a `su` no le pasamos como parámetro ningún nombre de usuario, asumirá que deseamos convertirnos en el administrador del sistema (*root*). Obviamente, si no conocemos la palabra clave del usuario, la orden fallará. La opción `-` se emplea para indicar a `su` que se tomen los parámetros de inicio (directorio de arranque, ruta de búsqueda de archivos, variables de entorno, etc.) definidos por el usuario al que nos convertiremos. Por defecto estos parámetros no se toman.

Ejemplo:

```
$ su - lucas
Password:
$ id
uid=519(lucas) gid=519(lucas) grupos=519(lucas)
$
```

newgrp

Sintaxis: newgrp [grupo]

La orden `newgrp` es similar a `su`, pero en este caso lo que se solicita es el cambio de identificador de grupo. Sólo nos podemos cambiar a los grupos permitidos por el administrador del sistema.

Ejemplo:

```
$ newgrp floppy
$ id
uid=504(chan) gid=19(floppy) grupos=504(igx)
$
```

2.6. Ejercicios

2.1 ¿Cuál es su directorio de arranque o directorio `HOME`? ¿Existe algún archivo oculto en su directorio de arranque? Haga un recorrido por los directorios más importantes del sistema visualizando los archivos contenidos en ellos.

2.2 Localice algún archivo ordinario, directorio, modo bloque y algún enlace simbólico.

2.3 Determine el tipo de los siguientes archivos: `/etc/hosts`, `/usr/bin`, `/etc/group`, `/bin/ls`, `/bin/login`, `/usr/lib/X11` y `/usr/include/stdio.h`.

2.4 Visualice las 7 primeras líneas y las 12 últimas del archivo `/etc/inittab`.

2.5 ¿Quién es el propietario del archivo `/etc/passwd`? ¿Y el grupo? ¿Cuántos enlaces tiene? ¿Cuál es la lista de derechos?

2.6 Cree en su directorio de arranque un subdirectorio denominado `copia` y copie en él el archivo `/etc/passwd`. ¿Quién es ahora el propietario del archivo? ¿Y cuál es su grupo?

2.7 Cambie el nombre del archivo `passwd` del directorio `copia` por el de `palabras_claves`.

2.8 Vaya al directorio `/etc` y cree un subdirectorio denominado `prueba`. ¿Qué ocurre? Compruebe los derechos que tiene en el directorio `/etc`.

2.9 Copie en su directorio de arranque un archivo cualquiera del directorio `/bin` y denomínelo `archivo1`. A continuación visualice el `archivo1` en formato largo. Haga un enlace del archivo anterior con un archivo denominado `nuevo`. ¿Cuántos enlaces tienen los archivos anteriores? ¿Es `nuevo` un archivo físico? ¿Qué ocurre si borramos el `archivo1`?

2.10 Vaya a su directorio de arranque, cree un subdirectorio denominado `.oculto.` ¿Qué ocurre si intenta visualizar el nuevo subdirectorio? ¿Qué opción debe emplear con `ls` para poder verlo? Copie en este directorio el archivo `/etc/hosts`. Visualice su contenido. Copie el archivo `/bin/cp` en el directorio `.oculto` que acaba de crear. Visualice el contenido de este archivo.

2.11 Mueva los archivos del directorio `.oculto` al directorio `copia`. ¿Qué archivos quedan en `.oculto`? Haga un enlace de los archivos que hay en `copia` al directorio `.oculto`. ¿Cuántos enlaces aparecen ahora por cada archivo? Borre los archivos de `copia`. ¿Cuántos enlaces aparecen ahora en los archivos de `.oculto`? Repita el proceso anterior, pero utilizando enlaces simbólicos.

2.12 ¿Puede cambiar el nombre de un directorio utilizando la orden `mv`? Compruébelo.

2.13 Cree un subdirectorio en su directorio de arranque denominado `tmp`. Copie en ese subdirectorio el archivo `/etc/group` con el nombre de `grupo`. Cambie los derechos de este archivo para que los usuarios de su grupo y el resto de los usuarios puedan modificarlo.

2.14 Cambie de propietario y de grupo al archivo `grupo` de su directorio `tmp`.

2.15 Elimine los tres subdirectorios que ha creado para realizar los ejercicios y compruebe qué ocurre.

2.16 ¿Qué valor deberíamos darle a la máscara de derechos para que todos los archivos se creasen con los atributos `rw-r--r--`?

2.17 ¿Cuáles son sus identificadores de usuario y de grupo?

2.18 Modifique sus identificadores de usuario y de grupo. ¿Qué utilidad tienen las órdenes anteriores?

Capítulo 3

El editor de texto `vi` (visual)

3.1. Qué es un editor

Un editor es una utilidad ofrecida por la mayoría de los sistemas operativos que nos permite modificar el contenido de un archivo. Cuando hablamos de editores o programas de edición, normalmente nos referimos a editores de texto; es decir, aquellos que trabajan con archivos que contienen cadenas de caracteres. Generalmente, los editores de texto son clasificados en dos categorías: los conocidos como editores de línea y los editores de pantalla. Un editor de línea es aquel en el que la unidad básica de trabajo es una línea o, lo que es lo mismo, una cadena de caracteres que termina con el carácter *newline* (`\n` en Linux). Un editor de pantalla nos permite visualizar una porción de un archivo (ventana de texto compuesta de varias líneas) en el terminal, así como que nos movamos con el cursor y efectuemos los cambios allí donde queramos.

Uno de los editores de texto más ampliamente utilizado en sistemas Linux es el editor de pantalla `vi` (visual), aunque `vi` sea un subconjunto de un editor mayor denominado `ex`. Este último incluye muchas más funciones y órdenes que el propio `vi`; sin embargo, raramente se utiliza. En un principio `vi` parece muy complicado de manejar, pero una vez que hemos practicado lo suficiente, veremos la potencia y la rapidez que posee. Un consejo práctico es que para aprender `vi` editemos textos. No por conocer todas sus opciones de memoria vamos a manejarlo mejor, lo más efectivo es practicar.

Cuando editamos con `vi`, trabajamos con una memoria intermedia (*buffer*); solamente cuando grabamos actualizamos el archivo en el disco. Son muchos los editores que hacen esto mismo, copiar el archivo inicialmente en una memoria intermedia y trabajar con él, porque tiene la ventaja de que si nos equivocamos podemos volver atrás sólo con salir sin grabar; de esa manera, el archivo inicial no se verá modificado. En contrapartida eso tiene el inconveniente de que si mientras estamos editando el sistema se viene abajo, los cambios hechos se perderán. Esta desventaja en el caso de Linux es menor, puesto que el sistema va haciendo a intervalos de tiempo una copia de esta memoria intermedia en el disco. Si cuando estamos editando el sistema cae, al arrancar de nuevo Linux nos enviará correo indicándonos cómo podemos recuperar dicho *buffer* perdido. Este método de utilizar un *buffer* también tiene la desventaja de que si el tamaño del archivo es mayor que el tamaño de la memoria intermedia, hay que dividirlo en partes para poder trabajar con él.

3.2. ¿Cómo podemos editar con vi?

Antes de invocar a vi, debemos asegurarnos de estar utilizando un terminal adecuado, ya que vi, como la mayoría de los editores de pantalla, necesita conocer el tipo de terminal para que funcione correctamente, de otro modo, los resultados pueden no ser los deseados. Para conocer el tipo de terminal, vi consulta al comenzar la variable de entorno TERM, y de esa manera, modifica la salida para que visualice el archivo eficazmente sobre el terminal. Nosotros podemos conocer el valor de esta variable del shell mediante la sentencia:

```
$ echo $TERM
vt100
$
```

la cual visualiza el valor de esta variable en ese instante. Si TERM no está iniciado a un valor correcto, podremos modificar su valor como indicamos a continuación. Suponiendo que nuestro terminal es ansi, para inicializar la variable de entorno de forma correcta haremos lo siguiente:

```
$ TERM=ansi
$ export TERM
$ echo $TERM
ansi
$
```

es necesario exportar la variable para que vi pueda acceder a ella. Si el lector quiere profundizar en el tema de las variables de entorno, deberá consultar el capítulo dedicado al shell. Si la variable TERM tiene ya un valor correcto, podremos comenzar a editar con vi dando la orden:

```
$ vi nombre_de_archivo
```

A partir de este momento, el archivo que queremos editar es copiado por vi en un *buffer*, la pantalla se borra y el cursor aparece localizado en el primer carácter de la primera línea del archivo. Si el archivo previamente no existía, vi lo creará (inicialmente vacío) con el nombre de archivo que le pasemos como argumento. Podemos también indicarle a vi desde la línea de órdenes que queremos que sitúe el cursor inicialmente al comienzo de una línea determinada del archivo; la forma de hacerlo sería:

```
$ vi +20 nombre_de_archivo
```

De esta manera, el cursor aparece ubicado inicialmente en el primer carácter de la línea número 20. Por último, si queremos que el cursor se sitúe al entrar en el primer carácter de la última línea, invocaremos a vi desde la línea de órdenes tecleando:

```
$ vi + nombre_de_archivo
```

De cualquier forma que llamemos a vi, éste nos ofrecerá una presentación similar a la siguiente:

```
$ vi carta
~
~
~
~
~
...
~
~
~
~
~
"carta" [New File]
```

Cuando vamos a editar un archivo nuevo, como ocurre en el ejemplo, el cursor inicialmente estará colocado en la primera línea y aparecerá parpadeante. El carácter ~ (tilde) indica que la línea está vacía, no contiene ningún carácter.

3.3. Estructura de las órdenes de `vi`

El editor `vi` tiene dos modos de trabajo, son los que se conocen con el nombre de modo mandato o modo orden y modo edición. Hablamos de modo edición cuando podemos introducir texto, y de modo mandato cuando `vi` nos permite dar órdenes propias de cualquier editor, líneas de texto, mover bloques, buscar palabras, etc. Inicialmente, cuando entramos en `vi`, éste se encuentra en modo orden, y por lo tanto no podremos introducir texto hasta que no introduzcamos la orden adecuada para ello. Una vez que nos hallemos en modo edición, para pasar a modo mandato deberemos pulsar la tecla de escape (ESC) situada en la parte superior izquierda del teclado. Las órdenes de `vi` tienen la siguiente expresión general:

```
{[}contador] operador {[}contador] objeto
```

La diferencia entre operador y objeto a veces no es nada evidente. Por ejemplo, la orden `w` avanza el cursor una palabra hasta el comienzo de otra, mientras que la orden `dw` borra la siguiente palabra del texto. En el primer caso, `w` actúa de operador, y en segundo, de objeto. El campo contador indica el número de veces que queremos repetir la operación. Este campo puede aparecer indistintamente en cualesquiera de los dos lugares en que aparece entre corchetes. Si aparece en los dos, el efecto será multiplicativo. Poniendo unos ejemplos, esta estructura de órdenes quedará más clara.

`w` Avanza una palabra hasta el comienzo de la otra.

`dw` Borra una palabra.

`3w` Avanza tres palabras.

`3dw` Borra tres palabras.

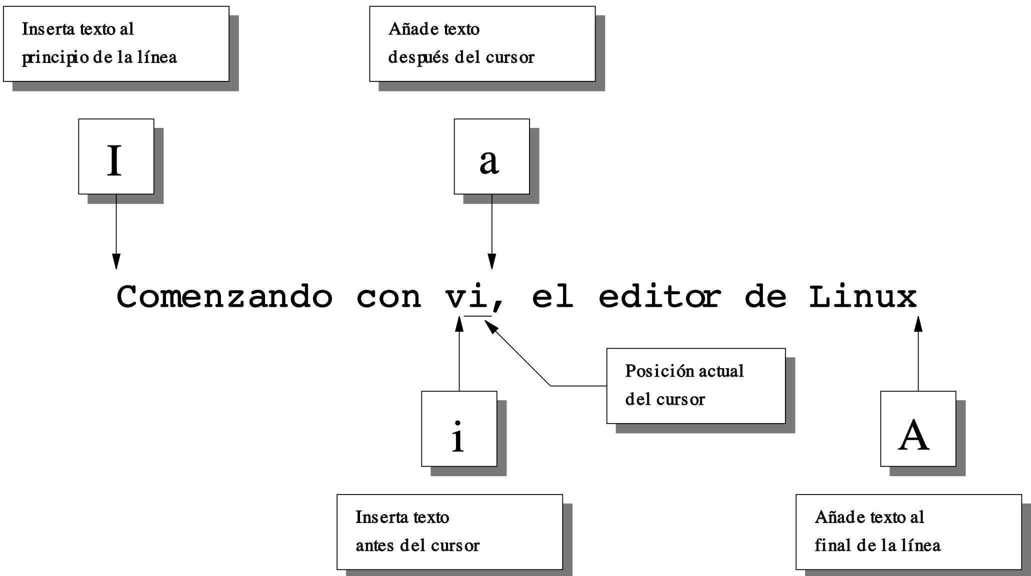


Figura 3.1: Órdenes básicas de `vi`.

`d3w` Borra tres palabras.

`3d3w` Borra nueve palabras (efecto multiplicativo, al aparecer el tres en los dos lugares).

3.4. Órdenes más comunes de `vi`

Para comenzar a escribir texto (pasar de modo mandato a modo edición), lo más común es utilizar una de las cuatro opciones que aparecen a continuación, y cuya explicación queda aún más clara en la figura 3.1.

`a` Añade (*append*) texto después de la posición del cursor.

`i` Inserta texto antes de la posición del cursor.

`A` Añade (*append*) texto al final de la línea.

`I` Inserta texto al principio de la línea.

Otras órdenes interesantes son:

`o` Abre la línea posterior de donde se encuentra actualmente el cursor.

`O` Abre la línea anterior de donde se encuentra actualmente el cursor.

`e` Avanza una palabra y el cursor queda colocado al final de la misma.

`b` Se mueve hacia atrás, hasta el principio de la palabra.

`dd` Borra la línea en la que está situada el cursor.

`U` Deshace el último cambio realizado en una línea.

`u` Deshace el último cambio.

`.` (Punto) Repite la última operación efectuada.

`x` Borra un carácter. Si a continuación pulsamos "`.`", repite el borrado.

`X` (*Backspace*) Borra caracteres hacia atrás.

`r` Reemplaza un carácter. Después de escribir el nuevo carácter, seguimos en modo mandato.

`R` Reemplaza caracteres (sobreescribir), no vuelve a modo mandato.

`ZZ` Salimos del editor guardando los cambios.

3.5. Movimientos del cursor

Para cambiar el cursor de situación, utilizaremos las teclas de cursor o, en su defecto, las teclas `h`, `j`, `k` y `l` (es fácil recordarlas porque están seguidas en el teclado).

`h` Cursor hacia la izquierda (←).

`j` Cursor hacia abajo (↓).

`k` Cursor hacia arriba (↑).

`l` Cursor hacia la derecha (→).

Existen otros modos para mover el cursor de forma más rápida, tales como:

`$` Mueve el cursor al final de la línea.

`^` Mueve el cursor al principio de la línea.

`H` (*Home*) Mueve el cursor al principio del texto de la ventana de texto.

`M` (*Middle*) Mueve el cursor a la mitad del texto de la ventana.

`L` (*Last*) Mueve el cursor al final del texto de la ventana.

3.6. Cambios de ventana

Todos los movimientos del cursor descritos se utilizan para movernos dentro de la ventana de texto ofrecida por `vi`. Existen órdenes que afectan al cambio de dicha ventana sin modificar la posición relativa del cursor en la pantalla. La orden `z` redibuja la pantalla, colocando la línea donde está situado el cursor en el medio, al principio o al final de la línea, dependiendo del carácter que siga a dicha orden.

`z <ENTRAR>` Coloca la línea donde se encuentra el cursor al principio de la pantalla.

`z.` Coloca la línea donde se encuentra el cursor en el medio de la pantalla.

`z -` Coloca la línea donde se encuentra el cursor al final de la pantalla.

Otras órdenes relacionadas con el cambio de la ventana son:

`^E` *Scroll up*, una línea.

`^Y` *Scroll down*, una línea.

`^D` *Scroll down*, media ventana.

`^U` *Scroll up*, media ventana.

`^F` *Forward*, avanza una página.

`^B` *Backward*, retrocede una página.

3.7. ¿Cómo salimos de `vi`?

Para salir de `vi` tenemos dos modos, el primero nos permite salir sin grabar, lo que haremos cuando nos hayamos equivocado y no queramos que el archivo original se vea afectado. Para salir sin grabar, desde modo mandato teclearemos:

`:q!<ENTRAR>`

Si al salir queremos guardar el archivo, desde modo mandato daremos cualquiera de las órdenes siguientes:

`:wq<ENTRAR>`

o

`:x<ENTRAR>`

Puede ocurrir que al querer salir grabando de `vi`, éste no nos permita guardar el *buffer*, porque al invocarlo estábamos situados en un directorio en el cual no tenemos derechos de escritura. Si nos vemos en un caso como el anterior, podremos indicarle a `vi` que grabe el archivo en un directorio en el cual sí tengamos la posibilidad de escribir. Un directorio que cumple este requisito puede ser el directorio de arranque del usuario, también conocido, como hemos indicado en otro punto, directorio HOME. La forma de hacerlo sería dando desde modo mandato la orden:

```
:w $HOME/nombre_del_archivo
```

y a continuación salir con la orden:

```
:q!
```

No hay problema al decir en este último caso que salimos sin grabar, puesto que previamente ya lo hemos hecho.

3.8. Opciones del editor

El editor `vi` tiene una serie de opciones accesibles por el usuario, el cual puede utilizarlas para personalizar en ciertos aspectos dicho editor. La forma de acceder a cada una de las opciones es teclear desde modo mandato

```
:set opcion
```

Con ello habilitaremos la opción deseada. Si posteriormente queremos desactivarla, también deberemos introducir desde modo mandato una orden del tipo

```
:set noopcion
```

El no delante de la opción deseada (y junto) provoca su anulación.

Para informarnos sobre el estado de todas las variables que pueden ser activadas o desactivadas, tendremos que usar la orden

```
:set all
```

De esta manera, `vi` nos informa sobre el estado de todas las opciones.

Ejemplos:

```
:set ai
```

Esta opción (*autoindent*) sirve, sobre todo, para facilitar la edición de programas. Si está habilitado, al pulsar ENTRAR el cursor no se vuelve a la columna cero, sino que se coloca alineado con el comienzo de la última línea. Para inhabilitar el *autoindent*, debemos dar la orden:

```
:set noai
```

Otra posibilidad interesante definible dentro de `vi` y muy usada también para la edición de programas es la opción conocida como *showmatch* o, escrita de modo abreviado, `sm`. Cuando esta opción está habilitada, cada vez que cerramos una llave, un paréntesis o un corchete, el cursor se coloca momentáneamente en la posición de la llave, paréntesis o corchete correspondiente, previamente abiertos (si se encuentran en la pantalla). Para activar la opción *showmatch*, debemos teclear desde modo mandato la orden:

```
:set sm
```

Para inhabilitarla, daremos la orden:

```
:set nosm
```

Para visualizar en todo momento el número de línea en la pantalla, debemos activar la opción `number` del siguiente modo (en forma abreviada)

```
:set nu
```

De esta manera, cada línea visualizada es precedida por su número correspondiente.

Si queremos que por defecto algunas opciones estén activadas al arrancar `vi`, debemos poner todas ellas en un archivo de configuración que lee el editor cuando lo invocamos. Dicho archivo reside en nuestro directorio `HOME` y se denomina `.exrc` (*ex run control*). Un ejemplo típico de archivo `.exrc` puede ser el siguiente:

```
$ cat .exrc
set autoindent autowrite showmatch report=1
set wrapmargin=8
$
```

3.9. Operaciones con palabras

Algunas de las operaciones más comunes con palabras son las comentadas en la lista siguiente:

`dw` Borra la palabra situada a continuación del cursor.

`cw` Cambia la palabra situada a continuación del cursor.

`D` Borra desde la posición del cursor hasta el final de la línea.

`C` Cambia desde la posición del cursor hasta el final de la línea.

`fa` Busca en la línea el carácter "`a`" (hacia adelante).

`Fa` Busca en la línea el carácter "`a`" (hacia atrás).

`;` Sigue buscando el mismo carácter en la misma dirección.

`,` Sigue buscando el mismo carácter en dirección contraria.

`J` Sirve para juntar líneas.

`G` Sirve para ir a la línea que le especifiquemos. Por ejemplo, `938G`, colocaría el cursor en la línea 938.

`dG` Borra hasta el final del archivo.

3.10. Órdenes más importantes en modo ex

Este modo, también denominado modo de última línea, se invoca desde modo mandato introduciendo `:` (dos puntos) y a continuación la orden `ex` deseada. Al hacer eso, el cursor se colocará en la última línea, y todo lo que tecleemos hasta pulsar la tecla ENTRAR será interpretado como una orden para el editor de línea `ex`. Ésta es la manera proporcionada por `vi` para acceder a órdenes de `ex`. Algunos ejemplos de órdenes de este tipo ya los hemos visto cuando explicábamos cómo salir de `vi` grabando o sin grabar. Veamos ahora otras capacidades del editor `ex` accesibles desde dentro de `vi`.

Para leer un texto procedente de un archivo o de una orden de Linux y cargarlo en la memoria intermedia, tenemos que utilizar la orden `r` (*read*) de `ex` seguida del nombre del archivo.

```
:r archivo
```

Lee el archivo `archivo` y lo carga en la memoria intermedia.

Si queremos editar un nuevo archivo vaciando la memoria intermedia actual, deberemos utilizar la orden `e` (*edit*) y a continuación el nombre del archivo.

```
:e archivo
```

Edita el archivo `archivo` vaciando la memoria intermedia actual.

En ocasiones quizá deseemos añadir el contenido actual de la memoria intermedia a un determinado archivo. En esos casos, debemos dar la orden:

```
:w >> archivo
```

Escribe el contenido de la memoria intermedia y lo añade al archivo `archivo`. El símbolo de redirección `>>` será explicado más profundamente cuando hablemos del shell.

Hay veces en que es necesario guardar en un archivo determinado parte de la memoria intermedia de edición. Para ello, disponemos de la orden siguiente, la cual escribe desde la línea `M` hasta la `N` de la memoria intermedia en el archivo especificado.

```
:M,Nw archivo
```

Escribe desde la línea `M` hasta la línea `N` desde la memoria intermedia al archivo `archivo`.

Para colocar el cursor en un determinado número de línea, tenemos que hacerlo de la forma siguiente:

```
:número
```

El cursor se va a la línea especificada en `número`.

Si, por ejemplo, tecleamos `:15`, el cursor se situará en la línea número 15. Esta orden es muy cómoda en el caso de que trabajemos en el desarrollo de software, porque si nos queremos situar en un determinado número de línea que nos ha indicado el compilador que contiene un error, lo haremos de una forma muy rápida.

Desde `vi` tenemos la posibilidad de ejecutar cualquier orden del shell sólo con teclear `:!` y a continuación la orden. Incluso desde el propio editor podemos lanzar un nuevo intérprete de órdenes, realizar las operaciones que deseemos y a continuación retornar a `vi` en el punto donde lo abandonamos simplemente tecleando Ctrl-d (ˆd) o `exit`.

Ejemplo:

```
:!sh
```

Con esto pasaremos a ejecutar un nuevo shell, y cuando estemos listos para retornar a nuestra sesión de edición, teclearemos `exit` o Ctrl-d, tal y como si fuésemos a desconectarnos del sistema.

3.11. Búsqueda de patrones

Dentro de `vi` tenemos la posibilidad de buscar una determinada palabra y colocar el cursor en la línea en la cual está situada. La forma de hacerlo es la siguiente:

```
/patrón
```

En este caso, busca en el texto el patrón especificado a partir de la posición del cursor hacia adelante. Si queremos buscarlo a partir de la posición del cursor hacia atrás, la forma de hacerlo sería:

```
?patrón
```

En cualquiera de los dos casos, si queremos repetir la búsqueda en la misma dirección que la búsqueda original, pulsaremos `n`, y si queremos hacerlo en dirección contraria, pulsaremos `N`. También tenemos órdenes que nos permiten buscar una determinada palabra y sustituirla por otra nueva, éstas son:

```
:1,$s /palabra_antigua/palabra_nueva/g
```

Cambia cada ocurrencia de `palabra_antigua` por `palabra_nueva` en toda la memoria intermedia.

```
:m,ns /palabra\_antigua/palabra\_nueva/
```

Cambia la primera ocurrencia de `palabra_antigua` por `palabra_nueva` desde la línea `m` hasta la `n`.

3.12. Marcas de posición

Cuando estamos editando un archivo con un tamaño muy grande, podemos marcar una posición determinada del archivo utilizando la orden `m` (*mark*) seguida de un carácter simple, el cual identificará dicha marca. Una vez puesta la marca, podemos retornar a ella simplemente tecleando el carácter ` (acento grave) y a continuación el nombre de la posición a donde queremos volver. Eso permite movernos de un lugar a otro dentro del archivo de una forma muy rápida. Como ejemplo podemos poner lo siguiente: `ma`, con lo cual incluimos una marca en la posición actual del cursor cuya etiqueta va a ser `a`. Si a continuación nos movemos con el cursor a otro lugar y posteriormente queremos volver al lugar original, deberemos teclear `` `a ``.

3.13. Mover bloques

Con el editor `vi` podemos copiar y mover bloques de texto de unas zonas a otras en el proceso de edición de un archivo. Para mover un bloque de un lugar a otro, colocaremos el cursor en la primera línea del bloque que queremos mover y a continuación borraremos con la orden `dd` el número de líneas que queramos trasladar. Por ejemplo, si damos la orden `10dd`, borraremos 10 líneas del texto; pero dichas líneas no son eliminadas definitivamente, sino que `vi` las lleva a un *buffer*. Posteriormente, colocaremos el cursor en el lugar donde decidamos colocar el texto borrado y pulsaremos `p` (*put*), con lo cual el *buffer* es restaurado en la nueva posición. Este procedimiento puede ser usado también para mover palabras o caracteres, pues al eliminar una palabra o un carácter, éstos son también llevados a un *buffer* auxiliar. El proceso de pegado del *buffer* puede repetirse tantas veces como sea preciso.

Para copiar bloques de texto, deberemos utilizar la orden *yank*, que nos permite llevar el texto a una memoria intermedia, pero manteniendo el texto inicial. Por ejemplo, si queremos llevar al *buffer* 5 líneas a partir de la posición actual del cursor, deberemos teclear `5yy` o `5Y`. Al hacer esto, `vi` mostrará un mensaje como el siguiente:

```
5 lines yanked
```

A continuación, para copiarlo, moveremos el cursor al lugar deseado y pulsaremos `p`. Los bloques también pueden ser guardados en *buffers* con nombre. Dicho nombre se compondrá de un solo carácter. Si queremos guardar 7 líneas en un *buffer* llamado `a`, deberemos teclear:

```
"a7yy
```

Con lo cual guardaremos 7 líneas en el *buffer* `a`. A continuación, para copiar el *buffer* en otro lugar, nos colocaremos con el cursor en la línea deseada, nombraremos el *buffer* y pulsaremos `p`.

```
"ap
```

Los *buffers* con nombre son mantenidos por `vi` aunque nos pongamos a editar otro archivo, siempre que no nos salgamos del editor. De esa manera, podremos copiar bloques de texto de unos archivos en otros.

3.14. Recuperación de archivos

Puede ocurrir que cuando estemos editando un archivo el sistema se venga abajo por un fallo de alimentación o que accidentalmente seamos desconectados. En estos casos, existe la posibilidad de recuperar el archivo que estábamos editando, incluso si no lo habíamos guardado. Si el archivo que perdemos tiene de nombre `tuberia.c`, la forma de recuperarlo sería la siguiente:

```
$ ex -r tuberia.c
```

Y de forma general:

```
$ ex -r nombre_archivo
```

3.15. La calculadora bc

Aunque este capítulo está dedicado al editor vi, con objeto de introducir algún texto de prueba para practicar con este editor, bc (*basic calculator*) que puede ser utilizado para realizar operaciones matemáticas. Esta calculadora puede operar de forma interactiva (leyendo en la entrada estándar) o bien procesar archivos que le pasemos como argumento. Estos archivos van a contener órdenes que son ejecutadas por la calculadora. La sintaxis de esta orden es la siguiente:

bc

```
Sintaxis: bc [-l] [-c] [archivo(s)]
```

-l Permite acceder a funciones de la biblioteca matemática.

-c No se invoca a dc, sólo se compila (realmente bc es un preprocesador que normalmente invoca a dc).

bc posee un lenguaje cuya sintaxis es muy similar a la del lenguaje C, posee identificadores, palabras reservadas, operadores y símbolos que serán descritos seguidamente. Antes de nada, vamos a poner un ejemplo de uso de la calculadora bc:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
123.132+75.64         orden
198.772               resultado
898.2345-34.23443     orden
864.00007             resultado
123*98                orden
12054                 resultado
5^10                  orden
9765625               resultado
1000/3                orden
333                   resultado
scale=10              orden
1000/3                orden
333.3333333333        resultado
sqrt(978212381237812) orden
31276386.9594589202   resultado
a=3.141592            orden
a*3                   orden
9.424776              resultado
quit                  orden
$
```

Inicialmente aparece una presentación que nos indica que la versión de `bc` que estamos utilizando ha sido desarrollada por la *Free Software Foundation*. Esta presentación no aparece en otras implementaciones de `bc`. Como podemos apreciar, con `bc` podemos hacer todo tipo de operaciones simples, pero, además, aporta operaciones más evolucionadas que veremos más adelante. Para terminar la sesión con `bc` daremos la orden `quit`. Aunque a primera vista `bc` parece una calculadora con poca potencia, la realidad es otra, ya que `bc` es capaz de llevar a cabo operaciones como las siguientes:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.

19723189739217398217983712897389217.123987128973982719378912 +
```

```
19723189739217398217983712897389217.123987128973982719378912 +
31290812098309218093801298309.123987213897231897321987       orden
19723221030029496527201806698687526.247974342871214616700899 resul
12^134                                                         orden
40764955294216304743794128079846299844235020571372407541675593946617\
72751249728065205173669089689216182244685486058202255169383625926645\
135704064                                                   resultado
sqrt(81923798127893279812378923718921793721987398)             orden
9051176615661263286317                                      resultado
quit
$
```

Realmente `bc` es una calculadora simbólica que permite llevar a cabo operaciones no realizables en las calculadoras ordinarias. Seguidamente vamos a citar los elementos del lenguaje de la calculadora `bc`.

3.15.1. Identificadores

Un identificador es un carácter simple perteneciente al intervalo `[a-z]` en minúsculas. Un identificador se utiliza para representar variables, matrices (*arrays*) y funciones. Dos identificadores idénticos no interfieren si representan distintos objetos; es decir, `x` como variable no tiene nada que ver con `x` como función.

Ejemplos:

`x` Variable `x`.

`x[i]` Elemento `i` de la matriz `x`. El rango de las matrices va desde 0 a 4097.

`x(a,b)` Función `x` con parámetros `a` y `b`.

3.15.2. Formatos de entrada-salida

Dentro de `bc` existen dos órdenes que nos permiten elegir la base del sistema de numeración que deseemos, tanto para el formato entrada de datos como para el de salida. Estas dos órdenes son:

`ibase = n` Indicamos que los números que introducimos desde el teclado están en base `n`. Por defecto, la base es 10.

`obase = n` La visualización de los resultados se hará en base `n`. También por defecto, `n` es igual a 10.

Otro punto que es posible definir en `bc` es el número de decimales con que se va a operar. La orden para definir este número de decimales es `scale`:

`scale = n` Los resultados se van a dar con `n` cifras decimales.

Vamos a poner un ejemplo en el que los números de entrada serán interpretados como números en binario. En este punto realizaremos una operación y el resultado será visualizado en decimal. A continuación haremos que los resultados se visualicen en octal y realizaremos la misma operación.

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
ibase=2      Números de entrada en binario
1001+0011    Operación
12           Resultado en decimal
obase=8      Números de salida en octal
1001+0011    Operación
14           Resultado en octal
quit
$
```

3.15.3. Palabras clave

Vamos a describir a continuación las palabras clave que se pueden utilizar en el programa `bc`:

if

```
if (expresión) {
    sentencias
}
```

Esta sentencia de control ejecuta las sentencias dependiendo de si la evaluación de expresión retorna un valor verdadero o falso. Las llaves solamente son necesarias cuando agrupamos varias sentencias dentro de `if`.

Ejemplo:

```
if (a == b) {
  x = x + a
  y = x + b
}
```

while

```
while (expresión) {
    sentencias
}
```

Las sentencias anteriores se repiten mientras la evaluación de expresión devuelva un valor cierto.

Ejemplo:

```
while (i < 20) a = a + i
```

for

```
for (v = e; condicion; progr\_cond) {
    sentencias
}
```

Esta sentencia de control se utiliza cuando deseamos repetir algunas sentencias un número determinado de veces.

`v = e` `v` representa la variable que será iniciada con el valor de `e`.

`condición` Representa la condición de mantenimiento dentro del bucle.

`progr_cond` Es una expresión cuyo valor evoluciona en el sentido que se dé a la condición para finalizar la ejecución de la sentencia `for`.

Ejemplo:

```
for (i = 0; i < 100; i++) a = a + 2
```

break

```
break
```

`break` se utiliza para finalizar cualquier bucle `for` o `while` aunque no se haya cumplido la condición de terminación.

3.15.4. Funciones

Es posible definir funciones dentro de `bc` con objeto de que puedan ser llamadas en cualquier momento. La forma de definir una función es la siguiente:

```
define f(x) {
    Cuerpo de la función
}
```

Aquí hemos definido una función denominada `f`, a la cual se le pasa como parámetro una variable que denominamos `x`. Es posible pasar varios argumentos a la función siempre que vayan separados por comas.

Si dentro de la función queremos utilizar variables propias de la función y que éstas no existan de forma global, deberemos declarar dichas variables en el cuerpo de la función de la siguiente manera:

```
c (a, b) {
    auto x
    x = a
    a = b
    b = x
}
```

La función anterior utiliza una variable denominada `x` que sólo existe dentro de la función `c`. Para indicar esto hemos hecho uso de la palabra reservada `auto`.

También podemos hacer que una función retorne valores, para lo cual debemos emplear la palabra reservada `return`.

Veamos un ejemplo. Supongamos que tenemos un archivo de texto donde está definida una función que interpretará `bc`, la cual calcula el cuadrado de un número. El contenido de este archivo es el siguiente:

```
$ cat cuadrado
define c(x) {
    auto a
    a = x^2
    return (a)
}
$
```

Ahora vamos a indicarle a `bc` que trabaje con este archivo, con lo cual dentro de la calculadora podremos utilizar la función indicada. Veámoslo:

```
$ bc cuadrado
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
n = c(3)
n        visualiza el valor de n
9
n = c(25)
n
625
quit
$
```

Como podemos observar, la función `c` calcula el cuadrado del número que le pasemos como argumento y devuelve el resultado con `return`.

En `bc` existen tres funciones predefinidas, además de las que se denominan funciones de biblioteca. Estas tres funciones son:

`sqrt(expresión)` Calcula la raíz cuadrada de `expresión`.

`length(expresión)` Calcula el número de dígitos de `expresión`.

`scale(expresión)` Calcula el número de dígitos decimales de `expresión`.

3.15.4.1. Funciones de la biblioteca matemática

Estas funciones que vamos a citar a continuación sólo son accesibles si ejecutamos `bc` con la opción `-l`.

`s(ángulo)` Calcula el seno del ángulo expresado en radianes.

`c(ángulo)` Calcula el coseno del ángulo expresado en radianes.

`a(x)` Calcula la arcotangente de `n` y devuelve el ángulo en radianes.

`e(expresión)` Calcula $e^{expresión}$.

`l(expresión)` Calcula el logaritmo de `expresión`.

`j(n,x)` Calcula la función de Bessel de orden `n`.

3.15.4.2. Operadores

Tenemos cuatro tipos de operadores: aritméticos, de asignación, relacionales y unarios.

- Aritméticos: + − ∗ % ^
- De asignación: = + = − = ∗ = / = % =^ =
- Relacionales: <= >= == ! =
- Unarios: − ++ −−

Para terminar, hay que decir que es posible poner comentarios dentro de `bc`, para lo cual se utilizan los siguientes símbolos:

```
/* Comentario */
```

Como ejemplo final, vamos a crear un programa que nos puede servir para calcular las soluciones de una ecuación de segundo grado. El programa lo vamos a denominar `2o_grado`, y su contenido es el siguiente:

```
$ cat 2do_grado
/* Resolución de una ecuación de 2º grado */
/* a b y c son los coeficientes del polinomio */

/* Visualiza este mensaje */
print "Ecuación de 2º grado"

a = 1
b = 7
c = 12

r = b^2-4*a*c
s = sqrt(r)

y = (-b+s)/(2*a)
z = (-b-s)/(2*a)

print "Solución 1:"
y

print "\n"
print "Solución 2:"
z

print "\n"
quit
$
```

Para procesar el archivo anterior, tendríamos que invocar a la calculadora `bc` del modo siguiente:

```
$ bc 2do_grado

bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
Ecuación de 2º grado
Solución 1:-3
Solución 2:-4
```

3.16. Ejercicios

3.1 Introduzca el siguiente texto con `vi` y guárdelo en un archivo denominado `ext2.doc`, colocado en un subdirectorio `doc` situado en su directorio de arranque.

```
                        El sistema de archivos de Linux
                       ---------------------------------


INTRODUCCIÓN


El sistema de archivos de Linux es la parte del núcleo (kernel)
encargada de gestionar los archivos del sistema. Entre sus funciones
podemos citar la creación y borrado de archivos y directorios, la
protección de la información, la lectura y escritura de datos, etc. Uno
de los objetivos planteados en su diseño es lograr la independencia de
dispositivo, de este modo las operaciones para acceder a los archivos
son siempre las mismas, independientemente de donde estén
localizados, disco, disquete o CD-ROM. Es más, el acceso a los
dispositivos de entrada y salida se realiza del mismo modo que el acceso
a archivos ordinarios.


CARACTERÍSTICAS DEL SISTEMA DE ARCHIVOS

El sistema de archivos de Linux tiene, cara al usuario, una estructura
en árbol invertido en el cual los archivos se agrupan en directorios. En
él, todos los archivos y directorios dependen de un solo directorio
denominado directorio raíz o root, el cual se representa por el símbolo
slash "/". En caso de que en el sistema tengamos varios dispositivos
físicos de almacenamiento secundario (normalmente discos o particiones
de disco), todos deben depender del directorio raíz y el usuario tratará
cada uno de los discos como un subdirectorio que depende de la raíz. A
esta operación se la conoce con el nombre de montaje de un subsistema de
archivos.

Los archivos se identifican en la estructura de directorios por lo que
se conoce como pathname o camino. Así la cadena /etc/passwd identifica a
passwd como un elemento que cuelga del directorio etc el cual a su vez
cuelga del directorio raíz (/). A partir de la cadena /etc/passwd no
podremos saber si passwd es un archivo o un directorio. Cuando el nombre
del camino empieza con el carácter / se dice que el camino es absoluto.
Linux también dispone de nombres de camino relativos, por ejemplo, si
nuestro directorio actual es /usr, la cadena bin/troff identifica al
archivo o directorio /usr/bin/troff. A esta cadena se la conoce, como
hemos señalado antes, como camino relativo puesto que no comienza con el
símbolo slash.
```