Introducción al Unix

Vicente González Ruiz (*vruiz@gogh.ualm.es*)

7 de abril de 1998

Capítulo 1

Lo básico de UNIX.

A mediados de los años 70, AT&T entregaba por 300 dólares los fuentes del sistema operativo UNIX a las universidades estadounidenses. En 1983 había más de 3000 centros académicos trabajando con ellos. Además existían libros que contenían el núcleo del sistema operativo comentado. Con toda esa información cualquier usuario podía detectar los fallos del sistema operativo o incluso modificarlo para ajustarlo a sus necesidades.

A principios de los 80, AT&T aplicó su derecho de propiedad sobre UNIX, el coste de los fuentes pasó a ser de 150000 dólares y se prohibió la difusión de los libros que comentaban el núcleo con el código fuente en C. Aún así, hoy en día pueden encontrarse libros publicados, que describen el núcleo a nivel de pseudocódigo.

Tal cantidad de información ha provocado que muchas personas (entre ellos Linus Torvalds) hayan trabajado con el sistema operativo. El resultado es que han aparecido fallos existentes y todo el mundo los ha conocido, por lo que se ha modificado el sistema operativo para resolverlos.

En este tema aprenderemos a usar los principales comandos del sistema operativo que usa el usuario normal (no el root). Todos estos programas de utilidad son comunes a todas las distribuciones de UNIX y las variaciones son mínimas: usaremos Linux para probarlos, pero podríamos haber usado cualquier otra versión del sistema operativo UNIX.

En gran cantidad de libros y manuales puede encontrar la historia y aventuras de cmo Ken Thomson ide un sistema operativo al que llam UNIX. Habr oido hablar tambin de SunOS, Solaris, Irix y AIX, y de cmo estos sistemas son los responsables de gran parte de la informtica actual (y no Microsoft, como errneamente se cree?), incluyendo Internet. Un hermano menor de todos ellos, Minix, era un sistema operativo ideado por A. Tanenbaum, que todo estudiante de informtica de los 80 tuvo que estudiar. En 1991, un estudiante finlands, a la sazn llamado Linus Torvalds comenz a crear un sistema operativo nuevo, basado en Minix, que le permitiese superar los numerosos inconvenientes del primero. Cada vez ms absorbido por su trabajo, el 5 de octubre de 1991 lanz a las news su famoso mensaje: "Echas de menos los maravillosos das de Minix 1.1, cuando los hombres eran hombres y escriban sus propios dispositivos? ?estoy trabajando en una versin gratuita de algo parecido a Minix para ordenadores AT-386. Ya he llegado a una fase en la que incluso se puede utilizar?". Era el comienzo del fenmeno Linux. Gran cantidad de programadores y usuarios, apasionados de la informtica, o simplemente desencantados con las prestaciones (y los precios y comportamientos) de los sistemas tradicionales, respondieron al desafo. En 1997, Linux es un competitivo, completo y altamente productivo sistema informtico, disponible para gran cantidad de tipos de ordenadores, que sigue respetando su filosofa original

Linux es una versin de UNIX, con unas caractersticas muy particulares. En primer lugar, es un sistema operativo. Adems incorpora cientos de aplicaciones y herramientas de productividad, desde compiladores y utilidades destinadas a los programadores, hasta un avanzado entorno grfico de usuario, X-Window, producto del trabajo del MIT durante muchos aos (no olvide que las ventanas y esas cosas son un invento de UNIX, no de Bill Gates, que slo las imit?). Todo ello pasando por editores de texto, multitud de juegos... No debe desdear las aplicaciones para X-Window, con herramientas que no tienen nada que envidiar a las existentes para Windows o Macintosh. Linux es, pues, un sistema informtico completo.

Linux es multiusuario. Es decir, que en un mismo ordenador pueden encontrarse trabajando al mismo tiempo, y con las mismas aplicaciones, gran cantidad de usuarios. Pueden tener diferentes privilegios, espacio reservado en disco para su trabajo, y acceder al mismo a cualquier hora y desde cualquier lugar del mundo, mediante una red de comunicaciones. Esto supone que alguien debe dedicarse a administrar el sistema, por lo que le resultar necesario aprender cosas que quedan muy lejanas a un usuario normal, pero que probablemente le sean tiles en un futuro.

Linux es gratuito. Cualquier usuario puede obtenerlo, instalarlo, copiarlo y distribuirlo. La mayor parte del software existente para Linux es gratuito: los programadores lo crean para solucionar sus necesidades, y luego lo ponen a disposicin del mundo, sin esperar recompensa econmica. Para salvaguardar los derechos de propiedad intelectual de los creadores del software, stos distribuyen sus creaciones bajo la licencia GNU, promovida y apoyada por la Free Software Foundation, que establece ciertas condiciones de copyright. Por supuesto, tambin puede encontrar shareware y aplicaciones comerciales para Linux, en las condiciones normales.

1.1 Arquitectura del sistema operativo UNIX.

Como todos los sistemas operativos UNIX gestiona los recursos físicos del ordenador (CPU, memoria, disco, terminales, etc) con las siguientes características:

1. Ser **multiusuario** permite que varios usuarios trabajen simultáneamente en el mismo ordenador, compartiendo así todos los recursos. En ocasiones alguien ha llamado a un sistema operativo de este tipo: multipuesto. Esto tiene una agradable ventaja: mucha gente puede usar simultáneamente la misma computadora y así aumentar la productividad. Probablemente si no fuera por esta cualidad del sistema operativo, no existirían grandes sistemas y todo el mundo debería trabajar con un PC.

2. Ser **multitarea** hace posible que un usuario pueda ejecutar simultáneamente varios programas. También se llama por esto multiprogramado. Realmente esta propiedad es casi una consecuencia de que el sistema operativo sea multiusuario, ya que la multitarea se puede conseguir dejando que un mismo usuario acceda al sistema más de una vez.

3. Ser en **tiempo compartido** posibilita que el sistema operativo distribuya controladamente los recursos del ordenador entre todos los usuarios. La cantidad de atención que un proceso recibe de la CPU puede ser modificado a antojo del usuario. A partir de UNIX System V Release 4, este sistema puede utilizarse para aplicaciones en tiempo real débil, es decir, aquellas cuyo tiempo de respuesta no puede garantizarse más que en términos estadísticos. Esto realmente es uno de los principales inconvenientes de UNIX para sistemas de tiempo real

en los que el tiempo de respuesta del sistema es una cuestión clave. Por ejemplo, UNIX no sirve para correr en la computadora de un avión que ejecute el código del piloto automático, ya que la probabilidad de respuesta inmediata no sería del 100%, es decir, un tanto por ciento de los aviones se estrellarían.

4. Ejecuta las aplicaciones según el modelo de **memoria virtual** que permite que un programa puede ejecutarse sin necesidad de que se almacene en su totalidad y simultáneamente en la memoria principal. Esto tiene tres grandes ventajas:

   (a) Puesto que sólo permanece el memoria la parte de un proceso que se está ejecutando, podemos almacenar muchos trozos de procesos simultáneamente (multitarea).

   (b) Un programa almacenado en disco y con un tamaño superior al de la memoria principal puede ejecutarse. Esto evita que los programas tengan que dividirse en *overlays* (partes disjuntas de un programa que si caben por separado en memoria principal) y que deben ser gestionadas directamente por el usuario.

   (c) Acelera el proceso de carga y lanzamiento de los procesos porque su ejecución puede comenzar en cuanto la primera página (bloque mínimo de transferencia entre el disco y la memoria principal) reside en memoria.

Tal y como se muestra en la figura 1.1, cuando un usuario entra a trabajar en un ordenador bajo UNIX le atiende un programa llamado **intérprete de comandos** (shell) que le permite ejecutar al menos 200 programas que forman parte de la distribución típica de UNIX. Sin embargo, desde un punto de vista técnico, el sistema operativo UNIX es tan solo el programa que gestiona los recursos del ordenador que se llama **núcleo** (kernel). Es por abuso del lenguaje por lo que habitualmente se habla de sistema operativo UNIX cuando nos referimos a cualquiera de los 200 comandos.

Quizás una de las principales características del sistema operativo UNIX es su asombrosa portabilidad a todo tipo de sistemas y la inalterable interfaz que muestra al usuario. Por esto, UNIX corre en la mayoría de las computadoras existentes (desde un IBM PC o CBM Amiga hasta un vectorial de Cray o Convex). En concreto lo que hace cada fabricante es adaptar UNIX y concretamente el núcleo a su máquina y de esta forma se comportará de forma similar a cualquier otra máquina que ejecuta UNIX. El secreto de esta facilidad de implantación se muestra en la figura 1.1 en la que se muestran los niveles que forman el sistema operativo UNIX. La característica más relevante de cualquier sistema UNIX es que existe un adecuado aislamiento entre cualquier usuario (incluido el superusuario) y el hardware del sistema, siempre usándose como intermediario el núcleo del sistema operativo que es la principal parte dependiente de la máquina. Por esta razón, Linux que no es más que una versión no comercial de UNIX (de ahí su bajo coste) corre actualmente en los siguientes sistemas (visitar `www.cs.us.es/archive/linux/devel.html` para más información):

1. **DEC AXP (Alpha)**. Es la última saga de máquinas de Digital que montan la familia de procesadores de 64 bits DEC Alpha 21064, 21064A y 21164 trabajan actualmente entre 150 y 633 MHz (por ahora sólo en máquinas monoprocesador). Para más información visitar `www.azstarnet.com/~axplinux`. Además ahora corre sobre `ARM`.

2. **Sparc**. Es la plataforma desarrollada por Sun Microsystems. Actualmente Linux corre sobre Sparc 1, 1+, 2, IPC, IPX, SLC, ELC, Sparc Classic, LX, 5, 10, 20, UltraSparc 140/170, y en breve en SparcCenter 1000/2000 y Sun 4/3xx. Para más información revisar `www.geog.ubc.ca/sparclinux.html`.

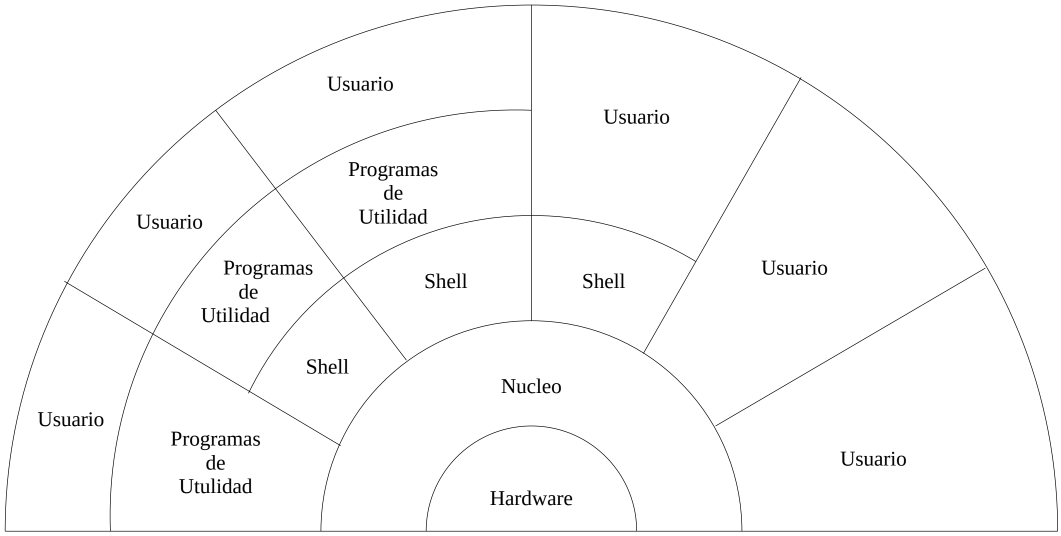

Figura 1.1: Arquitectura del sistema operativo UNIX.

3. **Power Macintosh**. Los nuevos Apple con CPU's PowerPC también corren Linux bajo el nombre de MkLinux. Concretamente Linux ha sido portado a Power Macintosh 6100, 7100, 8100 y PowerComputing 100 y 120. Información extra en `www.mklinux.apple.com`. También se está portando a Apple MAC con 680x0.

4. **IBM PowerPC**. También IBM corre Linux en sus nuevas estaciones de trabajo con PowerPC, concretamente en las series 830 y 850. Más información en `www.linuxppc.org`.

5. **Amiga**. Para el ya desaparecido CBM Amiga tenemos una versión de Linux llamada Linux/m68k que usa procesadores de la familia 680x0 (ahora hasta el 68060). Para más información atender a `ftp.informatik.uni-oldenburg.de/pub/amiga/linux`.

6. **Atari ST**. Esta máquina también monta un motorola 680x0 por lo que podemos encontrar información extra en la dirección de Amiga.

7. **Acorn**. Las máquinas `Acorn` pueden montar una tarjeta con un 486. Existe un Linux para esta configuración. Más información en `www.ph.kcl.ac.uk/~amb/linux.html`.

8. **MIPS**. MIPS es la empresa perteneciente a Silicon Graphics que fabrica los microprocesadores R2000, R3000, R4000, R44000, R5000, R8000 y R10000 (y alguna que otra variante más). También se montan en vídeo consolas como la PlayStation de Sony. Linux/MIPS corre actualmente en la subfamilia R4x00. Ejemplos de máquinas que montan este procesador son: `Acer PICA-61`, `DECStation 5000/2x`, `5000/100`, `3100`, `Deskstation rPC44`, `Mips Magnum 4000PC`, `Olivetti M700-10` y `RM200`.

9. **i386, i486, Pentium y Pentium Pro**. Originalmente Linux fue escrito para el i386 y este fue el principal acierto de Linus Torvalds pues escogió la plataforma más extendida en todo el mundo. Linux necesita únicamente 2 Mbytes (aunque tan poca memoria lo enlentece mucho) y un 80386SX para funcionar. Actualmente se está portando también a PC con un 8086 bajo el nombre de `Proyecto ELKS` (ver `www.linux.org.uk/Linux8086.html`) y a sistemas con

hasta 16 procesadores (Proyecto Linux/SMP). Un buen punto de partida para recavar información en castellano acerca de Linux es `slug.ctv.es/~alfredo`.

1.2 La primera sesión.

1.2.1 Entrando en el sistema (logging in).

Antes de comenzar a usar un sistema operativo compatible UNIX como es Linux, primero debemos de comunicar al sistema nuestra identidad. Para ello se nos pregunta por nuestro identificativo cuando aparece el mensaje de login, a lo cual nosotros teclearemos nuestro nombre en el sistema (normalmente una única palabra de al menos 6 caracteres, 2 de los cuales debieran de ser numéricos, con el fin de evitar intrusismos indeseados en el sistema de personas no autorizadas), y a continuación se nos pide la clave de acceso al sistema o password (si éste se ha establecido así, pues existen cuentas sin clave de acceso). Una vez tecleado convenientemente (y de forma secreta, pues no se realiza en eco en pantalla por razones de seguridad) nuestro password, el sistema operativo nos da la bienvenida. He aquí un par de ejemplos:

1. Entrando en una estación de trabajo SUN:

```
kua login: Isidoro
Password: (tecleamos nuestro password)
Last login: Wed Mar  8 02:09:00 on ttya
SunOS Release 4.1.1 (GENERIC_SMALL) #1: Thu Oct 11 10:48:40 PDT 1990
kua%
```

   Por este mensaje sabemos que la última vez que entramos en esta máquina fue el Miércoles 8 de Marzo a las 02:09:00 horas a través del terminal `ttya`. El sistema operativo es `SunOS` versión `4.1.1` (un poco viejo), el kernel o núcleo del sistema es del tipo `GENERIC_SMALL` y éste fue compilado por última vez el 11 de Octubre de 1990 a las 10:48:40. A continuación el `csh` (el intérprete de comandos) nos indica que está listo para atendernos interactivamente.

2. Entrando en un compatible PC con Linux:

```
Linux 1.0.8 (gogh....) (ttyp0)

gogh login: pepon
Password: (tecleamos nuestro password)
Last login: Mon Mar  6 08:43:06 from kua
Linux 1.0.8. (Posix).
gogh:~>
```

   En esta sesión podemos ver que la última vez que accedimos al sistema, lo hicimos desde kua (la estación de trabajo del ejemplo anterior). Es muy importante atender a esta información porque si hubiera un intento de acceso fallido a nuestra cuenta también sería indicado. Este se produce cuando alguien (incluido nosotros) intenta entrar en el sistema y falla el password después de dar nuestro login. Si el intento fallido se ha producido desde una máquina extraña, pudiera ser que alguien estuviera intentando colarse en la máquina usando nuestra cuenta, a lo que deberemos poner remedio cuanto antes.

1.2.2 Los shells de UNIX.

Los programas shell (de traducción concha) son intérpretes de comandos que facilitan el control del sistema operativo UNIX (son lo más parecido al `command.com` del `MS-DOS`). No son el sistema operativo en sí, sino que forman una capa que interacciona el núcleo del UNIX con el usuario interactivo. Al entrar al sistema, cada usuario corre un shell particular e independiente de cualquier otro usuario que simultáneamente esté utilizando el sistema. Esto se aprecia por la aparición de un carácter tal como `$` o `#` o `>` (existen otros más, de hecho ésto suele ser configurable) que dependerán del tipo de shell que hemos lanzado. Los shells son útiles para ejecutar uno o varios programas (simultáneamente entre sí y con el shell) y en ellos se aprecia la multitarea real del sistema operativo UNIX. Entre los shells más populares encontramos:

- **Bourne Shell**: fue escrito por Stephen Bourne y es el más básico y antiguo. Está disponible en todos los sistemas UNIX y se invoca a través del comando `sh` , tras lo que debe de aparecer el prompt del sistema `$`.
- **C Shell**: posterior al Bourne Shell y orientado a los programadores de C. También está disponible en todos los sistemas UNIX y se ejecuta escribiendo el comando `csh` , con lo que aparece el carácter `%` o `>`. Una versión mejorada de este es el `tcsh` que permite completar nombres de ficheros y directorios usando el tabulador y la edición de la línea de comandos. Este último suele estar disponible en todas las distribuciones de `Linux`.
- **Korn Shell**: escrito por David Korn, es una ampliación del Bourne Shell. Disponible en todos los sistemas UNIX, se lanza ejecutando el comando `ksh` , con lo que aparece el prompt `$`.
- **GNU Bourne-Again SHell (bash)**: este shell no aparece normalmente en las versiones comerciales del sistema operativo UNIX, aunque éste no es el caso en Linux. Como su nombre da a entender se trata de una extensión del Bourne Shell y añade características muy agradables tales como teclear trozos de nombres de ficheros por nosotros (al igual que `tcsh` y darnos listas de ficheros que empiezan por una determinada secuencia de caracteres sin mas que presionar la tecla de tabulación. Se invoca mediante el comando `bash` .

Ejemplos:

1. Ejecutando el Bourne Shell en SunOS:

```
Last login: Fri Mar 17 01:24:57 on ttyb (el sistema operativo nos indica el punto usado en el último acceso)
SunOS Release 4.1.1 (GENERIC_SMALL) #1: Thu Oct 11 10:48:40 PDT 1990
kua% sh (invocamos el Bourne Shell)
$
```

2. Ejecutando el C Shell en Linux:

```
Last login: Wed Mar 15 18:48:01 on tty3

Linux 1.0.8. (Posix).
gogh:~> csh (invocamos el C Shell)
>
```

3. Ejecutando el Korn Shell en Linux:

```
Last login: Wed Mar 15 18:49:55 on tty2

Linux 1.0.8. (Posix).
gogh:~> ksh (invocamos al Korn Shell)
$
```

4. Ejecutando el BASH Shell en Linux:

```
Last login: Wed Mar 15 18:55:29 on tty2

Linux 1.0.8. (Posix).
gogh:~> bash (invocamos al Bash Shell)
bash$
```

1.2.3 Control del terminal texto.

Caracteres especiales en UNIX.

Es conveniente tener en cuenta que UNIX es sensible a las mayúsculas (o a las minúsculas, como se desee entender). Además existen una serie de caracteres de control y secuencias de escape que siempre es necesario conocer. A estos caracteres a veces se les llama metacaracteres y pueden ser listados usando el comando `stty -a` (el comando es únicamente `stty` , `-a` es una opción, para obtener toda la información posible):

```
gogh:~> stty -a
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\ ; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
gogh:~>
```

El carácter ^ significa que deberemos de pulsar simultáneamente la tecla de control .
Las distintas combinaciones significan:

Combinación	**Significado**
`erase`	Retrocede con un espacio en blanco un solo carácter
`kill`	Borra la actual línea en el prompt del shell
`werase`	Borra la última palabra
`rprnt`	Reimprime la línea
`flush`	Ignora cualquier entrada pendiente y reimprime la línea
`lnext`	No trata el siguiente carácter de forma especial
`susp`	Suspende la ejecución de un proceso para despertarlo después
`intr`	Termina la ejecución de un proceso
`quit`	Termina la ejecución de un proceso y genera un fichero core
`stop/start`	Para/reinicializa la salida por el terminal
`eof`	El fin de fichero (End Of File), o también fin de la entrada

Terminando un proceso: ctrl - c .

Muy a menudo ocurre la situación de que lanzamos un proceso y nos arrepentimos antes de que este termine (si es que en algún momento va a acabar). La forma estándar (existen otras formas más refinadas y complejas) de hacer esto es pulsando simultáneamente las teclas ctrl y c , con lo que conseguiremos que el proceso que se está ejecutando superficialmente (foreground) y que bloquea el shell mientras no acaba, devuelva el control de la CPU. También diremos que existen procesos que no se pueden matar con esta secuencia de escape. Como ejemplo de aplicación:

```
kua% man chmod
Reformatting page.  Wait... done

CHMOD(1V)          USER COMMANDS            CHMOD(1V)

NAME
chmod - change the permissions mode of a file

SYNOPSIS
^C (terminamos el proceso de presentación de la ayuda)

kua%
```

Parando la salida por el terminal: ctrl - s / ctrl - q .

Si la salida de un proceso se produce rápidamente en tu terminal, y te es imposible leer la información que te está llegando, es posible detener el la presentación (scrolling) mediante la pulsación de las teclas ctrl y s simultáneamente. Para retomar el flujo de información teclear simultáneamente las teclas ctrl y q . A estas secuencias de control a menudo se les llama protocolo XON/XOFF.

Fin de la entrada: ctrl - d .

La mayoría de los programas de utilidad en UNIX pueden extraer sus datos de entrada de ficheros o desde el teclado de nuestro terminal. Quiere ésto decir que si llamamos a una determinada aplicación que espera que le indiquemos un fichero de entrada y no lo hacemos, aguardará a que este sea introducido por teclado. Lo mismo que todos los ficheros terminan en un carácter especial llamado EOF (fin de fichero), la secuencia de datos que estamos introduciendo por el teclado debe de acabar en un carácter especial de fin de entrada de datos. Este carácter se codifica mediante el uso simultaneo de las teclas ctrl y d . Como ejemplo, podemos usar la utilidad mail que nos permite escribir una carta (electrónica) a un colega nuestro:

```
gogh:~> mail isidoro
Subject: Hola isidoro

Que tal, isidoro !.
Bla, bla ....
Hasta pronto. Saludos.
(pulsamos ctrl - d )
Cc: (pulsamos enter )
gogh:~>
```

En este caso, también se nos está permitido acabar la carta introduciendo una línea en la que únicamente hay un punto.

1.2.4 Algunos programas de utilidad.

Para ejecutar un programa (que tenga o no utilidad), simplemente escribiremos su nombre y a continuación presionaremos la tecla $\boxed{\text{enter}}$. No todas las versiones de UNIX comparten los mismos programas de utilidad, pero si la mayoría.

Preguntando por la hora actual.

Por ejemplo, preguntar la hora con el comando $\boxed{\text{date}}$:

```
gogh:~> date (llamamos a la utilidad date )
Wed Mar 15 18:59:03 GMT 1995 (fecha y hora del sistema)
gogh:~>
```

O bien podemos limpiar nuestro terminal mediante el comando $\boxed{\text{clear}}$.

La ayuda ON-LINE del UNIX.

Como se ha podido apreciar, la descripción que se hace de los comandos más frecuentes es extremadamente básica. Debido a esto, solamente los flags de utilización y usos más comunes serán expuestos. Para saber más acerca de ellos se utiliza un comando muy útil. Todos los sistemas UNIX disponen de un programa de utilidad llamado $\boxed{\text{man}}$ (MANual page) que nos sirve para obtener información de cualquier comando estándar del sistema, de las instrucciones del lenguaje C (en la versión que soporta la computadora por defecto), de las estructuras de datos más comunes utilizada en el lenguaje C, y de los significados de ficheros. En realidad $\boxed{\text{man}}$ presenta al usuario la documentación original del sistema operativo UNIX tal y como viene en los manuales que normalmente son ocho volúmenes bastante extensos. Ocurre que no todos los manuales para las diferentes versiones de UNIX son idénticos. Podemos comprobar que dependiendo principalmente de si nuestra versión del sistema operativo desciende del System V o del BSD, que existen pequeñas variaciones en las opciones de los comandos. En nuestro caso Linux desciende del BSD, mientras que otros como el OSF/1 de Digital o el AIX de IBM descienden del System V. También se puede apreciar que no todas las opciones de los comandos expuestos en este documento están analizados. Sólo estudiaremos los más interesantes y usados. Tras esta pequeña aclaración, algunos ejemplos de la utilidad $\boxed{\text{man}}$:

1. Preguntando sobre el término `ports` (puertos) en SunOS:

```
kua% man -k ports
exports, xtab (5)       - directories to export to NFS clients
ext_ports (5)           - external ports file for network printers, terminals, and modems
kua%
```

2. Preguntando sobre la utilidad $\boxed{\text{chmod}}$ en SunOS:

```
kua% man chmod
Reformatting page.  Wait... done

CHMOD(1V)          USER COMMANDS          CHMOD(1V)

NAME
chmod - change the permissions mode of a file

SYNOPSIS
chmod [ -fR ] mode filename ...

SYNOPSIS
/usr/5bin/chmod [ -fR ] mode filename ...

AVAILABILITY       The System V version of this command is available  with  the
:
: (el resto de la ayuda)
:
CHMOD(1V)          USER COMMANDS          CHMOD(1V)

EXAMPLES
The first example denies write  permission  to  others,  the
second makes a file executable by all if it is executable by
anyone:

example          example
SEE ALSO
csh(1), ls(1V), sh(1), chmod(2V), chown(8)

Sun Release 4.1   Last change: 9 September 1987         3
```

3. Preguntando por la utilidad $\boxed{\text{man}}$ en bajo SunOS:

```
kua% man man
Reformatting page.  Wait... done

MAN(1)             USER COMMANDS              MAN(1)

NAME
man - display reference manual pages; find  reference  pages
by keyword

SYNOPSIS
man [-] [-t] [-M path] [-T macro-package]  [[section]  title
...] ...
man [-M path] -k keyword ...
man [-M path] -f filename ...

DESCRIPTION
man displays information from the reference manuals.  It can
:
:  (resto de la ayuda)
:
disadvantage of eliminating superscripts  and  subscripts  -
even  on those terminals that can display them.  CTRL-Q will
clear a terminal that gets confused by eqn(1) output.

Sun Release 4.1   Last change: 12 January 1988          4
```

Es una costumbre muy sana la de revisar el uso de un comando peligroso o poco frecuente, usando la utilidad $\boxed{\text{man}}$ antes de usarlo. Nos evitaremos de esta manera muchos problemas.

Contando el número de palabras que existen en un fichero texto.

La utilidad wc nos cuenta el número de líneas, palabras y caracteres de una lista de ficheros. Tomamos como palabra cualquier conjunto de caracteres separados por tabulaciones tab , espacios en blanco y caracteres de nueva línea. Si no se especifican ficheros de entrada, la entrada estándar se toma por defecto (el teclado). Su sintaxis:

wc *-lwc {fichero}**

La opción -l especifica que cuente número de líneas, la opción -w que cuente el número de palabras y la opción -c que averigue el número de caracteres. Si ninguna de las opciones es especificada, los tres cuenteos serán realizados. Ejemplo:

```
gogh:~/doc/Linux> wc Alumnos.txt
45     218    1535 Alumnos.txt
gogh:~/doc/Linux>
```

1.2.5 Estableciendo una clave de acceso al sistema.

Tras el primer `login` en un sistema multiusuario es recomendable modificar tu clave de acceso a tu cuenta, y normalmente repetir este proceso una vez al mes (aunque casi nadie hace caso de esto). Como ya hemos dicho, las claves de acceso (passwords) deberían tener al menos 6 caracteres y contener cifras numéricas. De esta forma nos aseguraremos de que la palabra no aparece en ningún diccionario, eliminado así la posibilidad de intrusismo en nuestra cuenta probando (de forma automática) con todas las palabras del diccionario. También deben de ser excluidos los nombres propios y apellidos. El comando que utilizamos para modificar el password es passwd , y aquí tenemos un ejemplo de aplicación en Linux:

```
gogh:~> passwd
Enter old password:      (introducimos el antiguo password)
Enter new password:      (tecleamos el nuevo password)
Re-type new password:    (retecleamos el nuevo password)
Password changed.
gogh:~>
```

Es importante no olvidar el `password` o de lo contrario deberemos contactar con el super usuario para que éste nos solucione el problema. Y un gran consejo: jamás olvideis la clave para el super usuario pues probablemente acabeis reinstalando el sistema operativo.

1.2.6 Saliendo de la cuenta.

Para abandonar el shell, lo más normal es teclear el comando exit o logout ,sin embargo algunos sistemas permiten salir de la cuenta con la secuencia de escape CTRL - D . La razón por la en ocasiones no permite por defecto esta forma de salir es por la proximidad de las teclas C , Z , ..., y D . A veces, nos salimos sin querer cuando deseábamos hacer otra cosa. Como veremos, esto depende de una variable de entorno. Valga como ejemplo:

```
gogh:~> logout   (pulsamos ctrl - d )

Welcome to Linux 1.0.8.

gogh login:(Linux aguarda un nuevo usuario)
```

1.3 Las cuentas de usuario.

Para que una persona pueda utilizar un ordenador corriendo UNIX, el administrador del sistema (root) debe abrirle una cuenta que físicamente se representa mediante un directorio perteneciente a dicho usuario. Para que éste pueda acceder a su cuenta, el administrador le asigna un nombre público (login) y una contraseña secreta (password). A partir de este momento, el usuario puede entrar en su cuenta a través de un terminal del ordenador. UNIX identifica a los usuarios que trabajan en la computadora cuando están registrados en el fichero `/etc/passwd` que contiene la siguiente información:

1. Nombre del usuario en el sistema (login).
2. La contraseña cifrada según el algoritmo DES [1] . A partir de UNIX System V Release 4, esta no aparece en el fichero `/etc/passwd` sino que lo hace en el fichero `/etc/shadow`. La razón de ésto es que el fichero `/etc/passwd` puede ser leído por cualquier usuario, así que cualquier usuario puede ejecutar un programa de descifrado y averiguar las contraseñas del resto de usuarios incluida la del superusuario (root). Sin embargo el fichero `/etc/shadow` sólo puede ser leído por el root.
3. El número identificador de usuario (User IDentificator: UID).
4. El número identificador de grupo al que pertenece el usuario (Group IDentificator: GID). En UNIX cada usuario pertenece al menos a un grupo de al menos un usuario. Todos los usuarios que pertenecen al mismo grupo pueden compartir sus ficheros. Los grupos creados en el sistema figuran en el fichero `/etc/group`.
5. Un comentario, que normalmente contiene el nombre y los apellidos del usuario.
6. Directorio de trabajo que representa la cuenta del usuario.
7. Programa que actúa como intérprete de comandos (shell).

Este es un ejemplo de fichero `/etc/passwd`. El carácter `:` separa los diferentes campos:

[1] En 1977 el National Boureau of Standard de EE.UU. (actualmente el National Institute of Standard and Technology: NIST) anunció un nuevo algoritmo criptográfico, que se denominó Data Encryption Standard: DES, para ser usado obligatoriamente en el cifrado de informaciones gubernamentales no clasificadas.

```
root:sV0m0suiSwxdk:0:1:system PRIVILEGED account:/root:/bin/csh
nobody:*Nologin:65534:65534:anonymous NFS user:/:
nobodyV:*Nologin:60001:60001:anonymous SystemV.4 NFS user:/:
daemon:*:1:1:system background account:/:
bin:*:3:4:system librarian account:/bin:
uucp:Nologin:4:2:UNIX-to-UNIX Copy:/usr/spool/uucppublic:/usr/lib/uucp/uucico
uucpa:Nologin:4:2:uucp adminstrative account:/usr/lib/uucp:
auth:*:6:11:Authentication Subsystem:/tcb/bin:
cron:*:7:14:Cron Subsystem:/usr/adm/cron:
lp:*:8:12:Line Printer Subsystem:/users/lp:
tcb:*:9:18:Trusted Computing Base:/tcb:
adm:*:10:19:Administration Subsystem:/usr/adm:
ris:Nologin:11:21:Remote Installation Services Account:/usr/adm/ris:/bin/sh
pepone:zoAgOqUKVtTcQ:305:10:Pepito Pepone Pepon:/home/pepone:/bin/csh
```

El siguiente listado es un ejemplo de fichero `/etc/group`.

```
system:*:0:root,pepone
daemon:*:1:daemon
uucp:*:2:uucp
mem:*:3:
kmem:*:3:root
bin:*:4:bin,adm
sec:*:5:
mail:*:6:mail
terminal:*:7:
tty:*:7:root
news:*:8:uucp
opr:*:9:root
auth:*:11:
lp:*:12:
lpr:*:12:root
backup:*:13:
cron:*:14:
users:*:15:pepone
sysadmin:*:16:
tape:*:17:
tcb:*:18:
adm:*:19:adm
operator:*:20:
ris:*:21:
```

En UNIX C1 (C1 es el nivel de seguridad que soporta el sistema operativo UNIX) existen dos tipos de cuentas:

1. **Superusuario (root)**. Posee todos los privilegios del sistema. Tiene su UID=0 y no se aplican restricciones a ninguna de sus acciones en el sistema.
2. **Usuario**. Es cualquier persona que no es root. Podrán realizar aquellas acciones que les permitan los ficheros a los cuales referencian.

Un usuario cualquiera puede realizar una sesión de trabajo en dos modalidades diferentes:

1. **Interactivamente**. Se conecta a su cuenta a través de un terminal y lanza comandos interactivos al sistema, esto es, esperando que el sistema le de respuesta más o menos rápidas.
2. **Mediante lotes**. El usuario puede lanzar tareas en el sistema en cualquier momento y sin requerirse su presencia en el terminal. Para ello se usa el comando $\boxed{\text{at}}$.

1.4 Tipos de ficheros en UNIX.

En UNIX existen 3 clases de ficheros diferentes: planos, directorios y especiales.

1.4.1 Ficheros planos.

Son los programas que contienen información generada durante la sesión de trabajo de los usuarios: documentos, fuentes de programas (por ejemplo en C o Fortran), datos, ejecutables, etc. Desde el punto de vista del sistema operativo un fichero plano o simplemente fichero es un conjunto consecutivo de octetos (bytes) sin estructura física alguna. Son los programas ejecutables que actúan sobre los ficheros quienes interpretan una estructura lógica sobre su contenido. Un ejemplo de fichero plano es el fichero `/etc/passwd`.

1.4.2 Ficheros directorios.

Los directorios son listas de ficheros que el sistema operativo manipula para conseguir una organización jerárquica arborescente. Por cada fichero o directorio que existe en un directorio existirá una entrada que contiene su nombre y un índice que permite acceder a su estructura de control asociada, que un UNIX se llama **nodo índice**. Un ejemplo de directorio es el `/etc`.

1.4.3 Ficheros especiales.

Cada dispositivo del ordenador (memoria, disco, terminal, diskette, ratón, etc) está representado por un fichero especial llamado fichero dispositivo. Existen dos tipos de dispositivos: (1) orientados a carácter y (2) orientados a bloque. Los primeros realizan sus operaciones de entrada/salida carácter a carácter (terminal, impresora, ratón, etc). Por el contrario, los segundos realizan sus operaciones en bloques de caracteres (discos, diskettes, etc). Todos los dispositivos soportados por un sistema operativo UNIX están contenidos en el directorio `/dev`.

1.5 Explorando el sistema de ficheros.

1.5.1 El sistema de ficheros.

El sistema de ficheros es una estructura lógica que utiliza el sistema operativo para manejar el conjunto de ficheros que hay en un disco, o en parte de él, y que desde el punto de vista del usuario se observa como un árbol cuya raíz está representada por el carácter $\boxed{/}$.

Todos los ficheros están ubicados dentro de algún sistema de ficheros (por ejemplo, el sistema de ficheros raíz puede montar a su vez un sistema de ficheros). El camino de un fichero es la ruta de nombres de directorios que debe recorrerse desde la raíz hasta el lugar en el que está ubicado el fichero. Como el camino suele ser muy largo, pueden aplicarse dos sinónimos para abreviar: (1) el punto $\boxed{.}$ representa al camino del directorio actual y (2) los dos puntos $\boxed{..}$ representan al camino del directorio padre del actual.

Todo ordenador que corre UNIX posee un sistema de ficheros que reconoce al arrancar el ordenador y que es el sistema de fichero raíz. En este normalmente los siguientes directorios:

1. `/bin`nary. Contiene los ficheros ejecutables que puede ejecutar un usuario del sistema.

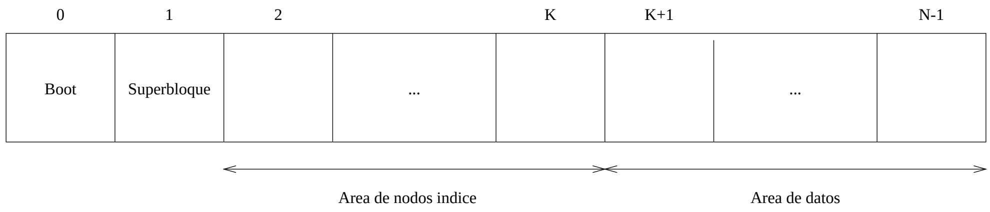

Figura 1.2: Formato interno de un sistema de ficheros.

2. `/device`. Contiene los dispositivos utilizables.

3. `/library`. Contiene los ficheros con código objeto (instrucciones de procesador no directamente ejecutables) de las bibliotecas para el compilador de C.

4. `/tmp`. Es el directorio usado para almacenar ficheros temporales. Cualquier usuario puede escribir en él y se borra periódicamente.

5. `/usr`. Contiene los ficheros y directorios relacionados con las aplicaciones de los usuarios.

6. `/usr/bin`. Más comandos de usuario.

7. `/usr/lib`. Bibliotecas de comandos.

8. `/usr/spool`. Colas de impresión, de mail, de procesamiento por lotes, etc.

Los discos pueden o no dividirse en particiones y en cada una de ellas se monta un sistema de ficheros independiente. El sistema de ficheros raíz normalmente se monta en la primera partición del primer disco y siempre está accesible. Para acceder a los demás sistemas de ficheros, el root los monta con el comando $\boxed{\text{mount}}$, en directorios vacíos del sistema de ficheros raíz o de otro directorio que cuelga del sistema de ficheros raíz. Sólo el root debe conocer sobre que disco se monta cada sistema de ficheros, si los discos son locales (están pinchados físicamente a la computadora) o son remotos (perteneces a otra computadora que corre UNIX y que ha exportado sus sistemas de ficheros) y de que tipo de disco se trata (diskette, disco magneto-óptico, CD-ROM, disco RAM, etc). El usuario normal únicamente ver un gran árbol de ficheros y directorios.

Internamente el sistema de ficheros se ve como un conjunto consecutivo de N bloques de disco (ver la figura 1.2). En el bloque 0 se ubica el programa para cargar el sistema operativo durante el arranque del ordenador (boot). El bloque 1 es el superbloque (superblock), que contiene información relativa al sistema de ficheros: tipo, tamaño, espacio libre, etc. Del bloque 2 al K se encuentra el área de nodos índice, y cada uno de éstos es un registro con información relativa a un fichero. Entre el bloque $K+1$ y $N-1$ están los bloques correspondientes al contenido de los ficheros.

Cada nodo índice (i-node) es un registro de 64 bytes que contiene la siguiente información acerca del fichero:

1. El identificador de usuario (UID) del creador (dueño) del fichero.

2. El identificador de grupo (GID) al que pertenece el fichero. Corresponde al grupo en el que se encontraba el dueño cuando creó el fichero.

3. Máscara de permisos (protecciones) del fichero. Estás definen las acciones que cualquier usuario puede realizar sobre el fichero.

4. Tipo de fichero: plano, directorio o dispositivo.
5. Tamaño en octetos del fichero.
6. Número de sinónimos del fichero. En UNIX, un fichero puede tener varios nombres. Estos se encuentran en sus directorios respectivos, pero todos ellos apuntan al mismo nodo índice. Esto es el número de enlaces que el fichero tiene.
7. Fecha de creación del fichero.
8. Fecha del último acceso (lectura o escritura) al fichero.
9. Fecha de la última modificación del nodo índice.
10. Tabla de apuntadores a los bloques de disco que constituyen el contenido del fichero. Esto no es aplicable a dispositivos.

1.5.2 Averiguando el directorio de trabajo.

UNIX lo mismo que otros sistemas operativos organizan sus sistemas de ficheros mediante un árbol de directorios que contienen ficheros. Tras el login, podemos encontrar nuestra situación en el árbol de directorios mediante el comando pwd (Print Working Directory). El directorio obtenido recibe el nombre de directorio trabajo (en el ejemplo siguiente, además coincide con el directorio de casa también llamado home directory). El `home directory` es aquel elegido por el administrador del sistema cuando nos creo la cuenta en el sistema. Un ejemplo de utilización del comando `pwd` podría ser:

```
gogh:~> pwd        (preguntamos por nuestro directorio de trabajo)
/usr/users/v       (ahora /usr/users/v es idéntico a ~)
gogh:~>            (como indica el prompt gogh:~>)
```

En UNIX existen caracteres especiales que identifican a ciertos directorios interesantes o muy utilizados, permitiendo un rápido acceso a ellos. El comando que nos permite acceder a un determinado directorio es cd (Change Directory). Los caminos hasta los directorios pueden ser absolutos (si es que lo designamos desde el directorio raíz) o relativos (respecto del directorio en el que nos encontramos). El la siguiente tabla aparecen como se designan estos directorios especiales:

Item	**Significado**
`.`	directorio actual
`..`	directorio padre al actual
`~`	directorio home
`/`	directorio raíz

El comando cd nos permite realizar lo siguiente:

`gogh:~> pwd`	*(vemos en directorio de trabajo)*
`/usr/users/v`	*(nuestro home directory es el actual work directory)*
`gogh:~> cd doc`	*(bajamos a un directorio llamado doc)*
`gogh:~/doc> cd`	*(volvemos a nuestro home directory)*
`gogh:~> cd bin`	*(bajamos a un directorio llamado bin)*
`gogh:~/bin> cd`	*(volvemos a nuestro home directory)*
`gogh:~> cd /`	*(subimos hasta el directorio raíz (root directory))*
`gogh:/> cd usr`	*(bajamos al directorio llamado usr que cuelga de /)*
`gogh:/usr> cd users`	*(bajamos a un directorio llamado users de camino absoluto /usr/users)*
`gogh:/usr/users> cd v`	*(bajamos al home directory de camino absoluto /usr/users/v)*
`gogh:~> cd ../..`	*(subimos hacia arriba dos directorios desde el home directory)*
`gogh:/usr> cd /usr/bin`	*(accedemos al directorio /usr/bin)*
`gogh:/usr/bin> cd ..`	*(subimos al directorio padre)*
`gogh:/usr> cd bin`	*(bajamos al directorio hijo llamado bin)*
`gogh:/usr/bin> cd`	*(volvemos a nuestro home directory)*
`gogh:~> cd .`	*(nos movemos al directorio actual work directory)*
`gogh:~>`	

1.5.3 Visualizando/creando un fichero texto.

La utilidad típica de visualización de ficheros texto cortos es cat (CATenate). Con ella es posible leer el contenido de un fichero ASCII en la pantalla del terminal texto. Su forma de uso es:

cat -n {*nombre_de_fichero*}*

Sin embargo las prestaciones de cat no acaban ahí, sino que aprovechando la redirección de ficheros del UNIX, nos permite crear ficheros ASCII o incluso añadir contenidos a uno ya creado. Posee una única opción llamada -n que permite añadir el número de línea a la salida. Por defecto, la entrada estándar es el teclado y la salida estándar es el monitor. La redirección de las entradas y salidas se realizan con los redirectores >, <, >>. Veamos algunos ejemplos:

```
gogh:~> cat > un_fichero_ASCII
Un fichero ASCII contiene bytes que pertenecen todos a la parte de la tabla
ASCII que generan caracteres normales.
Para terminar de introducir información pulsamos ^D.
gogh:~> ls -l un_fichero_ASCII
-rw-rw-r--   1 vi       wheel         171 Mar 15 20:17 un_fichero_ASCII
gogh:~> cat un_fichero_ASCII
Un fichero ASCII contiene bytes que pertenecen todos a la parte de la tabla
ASCII que generan caracteres normales.
Para terminar de introducir información pulsamos ^D.
gogh:~> cat >> un_fichero_ASCII
y ahora añadimos una linea mas, tras pulsar de nuevo ^D.
gogh:~> ls -l un_fichero_ASCII
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~> cat un_fichero_ASCII
Un fichero ASCII contiene bytes que pertenecen todos a la parte de la tabla
ASCII que generan caracteres normales.
Para terminar de introducir información pulsamos ^D.
y ahora añadimos una linea mas, tras pulsar de nuevo ^D.
gogh:~> cat < un_fichero_ASCII
Un fichero ASCII contiene bytes que pertenecen todos a la parte de la tabla
ASCII que generan caracteres normales.
Para terminar de introducir información pulsamos ^D.
y ahora añadimos una linea mas, tras pulsar de nuevo ^D.
gogh:~>
```

1.5.4 Listando los contenidos de un directorio.

Los ficheros y directorios que contiene un directorio pueden listarse (a través de la salida estándar) con el comando $\boxed{\text{ls}}$, que puede utilizarse con las siguientes opciones de funcionamiento:

ls *-adglsFR*{*nombre(s) de fichero(s)*}{*nombre(s) de subdirectorio(s)*}

donde ningún argumento listará todos los archivos del subdirectorio en orden alfabético excluyendo aquellos que comienzan por un punto (.), que son los llamados ficheros ocultos. Como ejemplo:

```
iron% ls
1           carta      core          f           mpi
bat         carta2      dead.letter  img
bin         carta2%    doc          mbox
iron%
```

La opción $\boxed{\text{-a}}$ causa que todos los ficheros se incluyan en el listado:

```
iron% ls -a
.                              .openwin-init
..                             .wastebasket
.Xauthority                    1
.Xdefaults                     bat
.Xdefaults.old                 bin
.chimpv2rc                     carta
.cshrc                         carta2
.desksetdefaults               carta2%
.gopherrc                      core
.login                         dead.letter
.mailtool-init                 doc
.mosaic-global-history         f
.mosaic-hotlist-default.html   img
.mosaic-personal-annotations   mbox
.mosaicpid                     mpi
iron%
```

donde aparecen nuevos ficheros ocultos tales como:

Fichero	Significado
`./`	directorio de trabajo.
`../`	directorio padre.
`.cshrc`	script lanzado cuando arranca el C Shell.
`.login`	script lanzado cuando accedemos al sistema.

La opción $\boxed{\text{-d}}$ selecciona el directorio de trabajo (su utilidad parece ser algo oscura):

```
iron% ls -d
.
iron%
```

La opción $\boxed{\text{-l}}$ presenta información acerca de los permisos, el número de enlaces físicos hardware y software que ocupa la entrada (fichero o directorio), el dueño, el tamaño en bytes, la fecha de la última modificación y el nombre de la entrada:

```
iron% ls -l
total 315
lrwxrwxrwx  1 vi             21 Mar  8 18:44 bat -> /home1/paralelo/v/bat
lrwxrwxrwx  1 vi              9 Mar 21 19:20 bin -> /home/bin
-rw-rw-r--  1 vi           5408 Apr  4 12:25 carta
-rw-rw-rw-  1 vi            380 Apr  5 17:59 carta2
-rw-rw-r--  1 vi            346 Apr  4 12:51 carta2%
-rw-r--r--  1 vi        8577456 Apr  5 12:01 core
-rw-------  1 vi              0 Apr  6 20:03 dead.letter
lrwxrwxrwx  1 vi             21 Feb 15 14:00 doc -> /home1/paralelo/v/doc
lrwxrwxrwx  1 vi             19 Feb 15 14:00 f -> /home1/paralelo/v/f
lrwxrwxrwx  1 vi             21 Mar 22 14:12 img -> /home1/paralelo/v/img
-rw-------  1 vi          28742 Apr 10 10:13 mbox
drwxrwxr--  3 vi            512 Feb 17 09:59 mpi
iron%
```

La opción $\boxed{\text{-g}}$ presenta el grupo, cuando este no es enseñado por la opción $\boxed{\text{-l}}$:

```
iron% ls -lg
total 315
lrwxrwxrwx  1 vi       staff          21 Mar  8 18:44 bat -> /home1/paralelo/v/bat
lrwxrwxrwx  1 vi       staff           9 Mar 21 19:20 bin -> /home/bin
-rw-rw-r--  1 vi       staff        5408 Apr  4 12:25 carta
-rw-rw-rw-  1 vi       staff         380 Apr  5 17:59 carta2
-rw-rw-r--  1 vi       staff         346 Apr  4 12:51 carta2%
-rw-r--r--  1 vi       staff     8577456 Apr  5 12:01 core
-rw-------  1 vi       staff           0 Apr  6 20:03 dead.letter
lrwxrwxrwx  1 vi       staff          21 Feb 15 14:00 doc -> /home1/paralelo/v/doc
lrwxrwxrwx  1 vi       staff          19 Feb 15 14:00 f -> /home1/paralelo/v/f
lrwxrwxrwx  1 vi       staff          21 Mar 22 14:12 img -> /home1/paralelo/v/img
-rw-------  1 vi       staff       28742 Apr 10 10:13 mbox
drwxrwxr--  3 vi       staff         512 Feb 17 09:59 mpi
iron%
```

En este caso el grupo es `staff`. La opción $\boxed{\text{-s}}$ nos dice en número de bloques de disco (típicamente 512 bytes/bloque) que ocupa cada entrada en el directorio:

```
iron% ls -s
total 315
1 bat              1 carta2          0 dead.letter     1 img
1 bin              1 carta2%         1 doc            29 mbox
6 carta          272 core            1 f               1 mpi
iron%
```

Como puede observarse, no se incluye el espacio ocupado por los ficheros o directorios que tenga que cuelguen de un directorio. La opción $\boxed{\text{-F}}$ causa que después del nombre de cada entrada en el directorio aparezca un carácter que indica su naturaleza:

```
iron% ls -F
bat@         carta        carta2%      dead.letter  f@           mbox
bin@         carta2       core         doc@         img@         mpi/
1*
iron%
```

Los sufijos que podemos obtener son:

Símbolo	**Significado**
`@`	La entrada es un enlace simbólico.
`/`	La entrada es un directorio.
`*`	La entrada es un fichero ejecutable

Por último, la opción -R causa que el comando se realice recursivamente para cada directorio. El comando ls es probablemente el más usado de todos ya que nos aporta mucha información. Supongamos que hemos construido la siguiente entrada:

```
iron% ls -algFs doc
1 lrwxrwxrwx  1 vi       staff          21 Feb 15 14:00 doc -> /home1/paralelo/v/doc/
iron%
```

En este ejemplo, podemos extraer la siguiente información:

Campo	Valor	Significado
1	`1`	Número de bloques usados en el almacenamiento físico del fichero
2	`lrwxrwxrwx`	Permisos sobre el fichero para los distintos usuarios
3	`1`	Número de enlaces hard asociados al fichero
4	`vi`	Identificativo del usuario propietario del fichero
5	`staff`	Identificativo del grupo propietario del fichero
6	`21`	Tamaño del fichero en bytes
7	`Feb 15 14:00`	Fecha de la última modificación
8	`doc`	Nombre del fichero o directorio

Cada uno de estos campos significa lo siguiente:

1. **Almacenamiento del fichero.** El primer campo indica el número de bloques de disco que el fichero ocupa físicamente. Esto es útil cuando trabajamos con ficheros dispersos los cuales en el campo sexto pueden arrojar un tamaño muy alto en bytes y sin embargo ocupar pocos bloques de 512 bytes.

2. **Permisos del fichero o directorio.** En el segundo campo describimos los permisos sobre el fichero o directorio. Está compuesto de 10 caracteres, cuyo significado es:

   1: Indica el tipo de fichero:

Caracter	Tipo de fichero
`-`	ficheros normales.
`d`	directorios.
`b`	ficheros especiales buffereados (discos duros, CD-ROM's, ...).
`c`	ficheros especiales no buffereados (terminales,...).
`l`	enlace simbólico.
`p`	pipe.
`s`	socket.

   2,3,4: Permisos para el propietario del fichero. Responden a la siguiente tabla:

permiso	fichero regular	directorio	fichero especial
`r`	leerlo	listar su contenido	extraer datos del dispositivo
`w`	escribirlo	modificar su contenido	escribir datos en el dispositivo
`x`	ejecutarlo	acceder subdirectorios	sin significado

   5,6,7: Permisos para el grupo de trabajo al que pertenece el propietario.

   8,9,10: Permisos para cualquier otro usuario del sistema.

1.5.5 Creando enlaces hard y enlaces simbólicos.

La utilidad ln te permite crear dos tipos de enlaces a tus ficheros y directorios. Un enlace es un nombre alternativo que podemos asignar a un fichero o a un directorio ya existente. La finalidad de esto es el realizar cómodamente el acceso a dichos ficheros o directorios sin necesidad de referenciarlos con su camino completo a través del sistema de ficheros. Por ejemplo, supongamos que en nuestro lugar de trabajo tenemos varias estaciones de trabajo bajo UNIX y que una de ellas es un servidor de servidor de ficheros. Lo más normal es que cada estación realice un `login` en su disco disco local. Si somos programadores interesa que nuestros programas fuente no estén duplicados en cada disco local, ya que además de desperdiciar espacio, deberíamos de recordar sobre que máquina estuvimos trabajando la última vez y copiar los fuentes sobre la estación de trabajo de ese día. UNIX permite compartir sistemas de ficheros de forma que una estación exporta su disco local (o parte de él) para que otras estaciones lo puedan ver como un disco local. Una vez que es posible acceder a un disco disco remoto (montado) se puede crear un enlace simbólico desde nuestro home directory en la estación sobre la que estamos trabajando ha cia el camino completo a través del sistema de ficheros, donde se encuentran nuestros programas fuente. Si realizamos este proceso en cada estación de trabajo a la que tenemos acceso, parece que en cada `home directory` de cada máquina hay un directorio local que contiene los fuentes, cuando lo que realmente existe en un enlace simbólico desde cada uno de los `home directory` sobre el sistema de ficheros soportado por el servidor de ficheros. Por ejemplo:

```
gogh:~> pwd
/usr/users/v     (nuestro home directory)
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~>
```

donde hay dos enlaces simbólicos creados en nuestro `home directory` sobre un disco remoto montado sobre `gogh` como `home1`. La diferencia principal existente entre un enlace simbólico y un enlace físico es que este segundo no puede establecerse entre sistemas de ficheros. La utilidad que crea enlaces es:

ln *-sf nombre_de_fichero_o_directorio [nombre_del_enlace]*

Como ya hemos dicho, la opción -s crea enlaces simbólicos a través de sistemas de ficheros (distintas particiones de un mismo disco o discos distintos). Como ejemplo:

```
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~> ln -s /home1/paralelo/v/cintas music
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
lrwxrwxrwx   1 vi       wheel          23 Apr  8 23:11 music -> /home1/paralelo/v/music/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~>
```

Es posible borrar un enlace físico o simbólico sin que se borre físicamente la información sobre el sistema de ficheros:

```
gogh:~> rm music
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~> ls -l /home1/paralelo/v/cintas
total 14
-rw-rw-rw-   1 vi       wheel        7068 May 18  1994 cintas
gogh:~>
```

1.5.6 Modificando los permisos a un fichero o directorio.

Se usa el comando chmod . El super usuario puede modificar los permisos de cualquier fichero regular o directorio, sin embargo, el usuario normal sólo podrá alterar los permisos de aquellos ficheros sobre los que previamente tenga el permiso de escritura. Su sintaxis sería:

chmod *-R cambio {,cambio}*{fichero}+*

donde *cambio* está formado por una de las siguientes formas:

selección_de_cluster	+	nuevos_permisos (añadir permisos)
selección_de_cluster	-	nuevos_permisos (eliminar permisos)
selección_de_cluster	=	nuevos_permisos (asignar permisos absolutamente)

donde la selección_de_cluster es cualquier combinación de:

Caracter	Significado
u	usuario/propietario
g	grupo
o	otros
a	todos

y donde los nuevos_permisos es cualquier combinación de:

Caracter	Significado
r	lectura
w	escritura
x	ejecución/acceso a directorio
s	activar el identificativo de usuario/grupo durante la ejecución
X	ejecutar sólo si el fichero es un directorio o si algún usuario tiene permisos de ejecución
t	grabar el segmento de texto del programa en el dispositivo de swapping
u	los permisos que el usuario tiene actualmente
g	los permisos que el grupo tiene actualmente
o	los permisos que el resto de usuarios tiene actualemten

contenidos en todos los directorios que cuelgan del indicado en el comando. Veamos varios ejemplos:

```
gogh:~> ls -lg
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~> chmod o-r un_fichero_ASCII   (eliminamos el permiso de lectura para otros)
gogh:~> ls -lg
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw----   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~>
gogh:~> chmod 664 un_fichero_ASCII   (dejamos el fichero como estaba)
gogh:~> ls -la
total 12
drwxr-xr-x   3 vi       wheel        1024 Apr  8 23:13 ./
drwxrwxrwx   3 root     root         1024 Feb 14 01:47 ../
-rw-r--r--   1 vi       wheel         225 Mar 15 18:55 .bash_history
-rw-r--r--   1 vi       wheel        1440 Feb 14 02:09 .cshrc
-rw-r--r--   1 vi       wheel        2805 Feb 14 01:50 .login
-rw-rw-r--   1 vi       wheel          44 Mar 15 18:57 .pdksh_hist
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~>
```

1.5.7 Averiguando el tipo de un fichero.

Podemos saber el tipo de un determinado fichero usando el comando:

file *-L* {*nombre_de_fichero*}+

La opción -L hace que cuando indiquemos el fichero a través de un enlace simbólico, se nos describa el contenido del fichero y no del enlace simbólico. Ejemplos:

```
gogh:~> file *
bin:              directory
carta             English text
doc:              symbolic link to /home1/paralelo/v/doc
f:                symbolic link to /home1/paralelo/v/f
un_fichero_ASCII: data
gogh:~> file -L *
bin:              directory
carta:            English text
doc:              directory
f:                directory
un_fichero_ASCII: data
gogh:~>
```

1.5.8 Otras formas de visualizar el contenido de un fichero.

Además de la utilidad $\boxed{\text{cat}}$ anteriormente descrita, podemos ver el contenido de grandes ficheros con las utilidades $\boxed{\text{more}}$ y $\boxed{\text{page}}$. Sus funcionamientos son:

more *[+ número de línea]*{*nombre de fichero*}

donde ninguna opción posibilita presentar el contenido de un fichero ASCII página por página (pulsando intermitentemente la tecla de $\boxed{\text{space}}$) o línea a línea (pulsando la tecla de $\boxed{\text{return}}$). Pulsando la tecla $\boxed{\text{q}}$ abortamos el proceso y pulsando la tecla $\boxed{\text{h}}$ obtenemos un listado de otras posibles opciones de funcionamiento interactivo.

page *[+ número de línea]*{*nombre de fichero*}

la única diferencia con $\boxed{\text{more}}$ está en la forma de presentar el pantalla los contenidos de los ficheros. $\boxed{\text{more}}$ realiza scrolling vertical (desplazamiento de las líneas de la pantalla, una a una), mientras que $\boxed{\text{page}}$ borra la pantalla antes de presentar la siguiente, característica que en ocasiones (como es en el caso de las comunicaciones a través de módem) es de agradecer por el aumento en la velocidad de presentación. Cuando lo que nos interesa es averiguar de que va el contenido de un fichero, nada más útil que poder echar un vistazo al principio o al final del mismo. Para estas acciones se han creado las utilidades $\boxed{\text{head}}$ y $\boxed{\text{tail}}$, respectivamente. He aquí sus modos de empleo:

head *-n* {*nombre de fichero*}

tail *-n* {*nombre de fichero*}

donde la $\boxed{\text{-n}}$ indica el número de líneas de fichero ASCII presentadas (por defecto diez). Ejemplos:

```
gogh:~/doc/Linux> more alumnos.txt      (page funciona igual)
1  Periquillo de los Palotes, Pepe
2  Agujero Solo, Bartolo
:
:     unos cuantos más
:
22  Gil Puertas, Jesús
23  Estroncio Deuterio, Cerio
--More--(51%)    (aquí espera a que pulsemos [space] pero pulsamos la [q] )

gogh:~/doc/Linux> head alumnos.txt      (los primeros 10 alumnos del curso)
1  Periquillo de los Palotes, Pepe
2  Agujero Solo, Bartolo
:
:     unos cuantos más
:
9  Pi, Filemon
10  X, Mortadelo

gogh:~/doc/Linux> tail alumnos.txt      (los últimos 10)
36  Ritchie, Dennis
37  Kernighan, Brian
:
:     unos cuantos más
:
43  Gimenez, Curro
44  Knuth, Donald
45  Torvalds, Linux
gogh:  /doc/Linux>
```

Existe una versión mejorada de `more` llamada `less` (que en inglés significa menos, lo contrario de *more*). Su principal ventaja radica en que no necesita leer todo el fichero de entrada tal y como hace `more` antes de presentar el contenido de este por pantalla. Por esta razón, `less` es más rápido para visualizar grandes ficheros.

1.5.9 Moviendo/renombrando un fichero o directorio.

La utilidad para realizar el movimiento de ficheros o directorios a través del árbol de directorios del sistema de ficheros se llama `mv` . Cuando no referenciamos a dos sistemas de ficheros distintos con el comando `mv` , no se produce el movimiento físico (byte a byte) de los ficheros (o directorios con sus contenidos) sino que se realiza una modificación de las etiquetas de la jerarquía. Esta es la razón por la cual a veces nos sorprenda la velocidad con la que `mv` trabaja. Podemos realizar tres movimientos distintos[2]

- ◇ **mv** *[-] -i antiguo_fichero nuevo_fichero*
- ◇ **mv** *[-] -i {fichero}* directorio_destino*
- ◇ **mv** *[-] -i antiguo_directorio nuevo_directorio*

con los que renombramos un fichero (si *nuevo_fichero* existe es reemplazado, ¡ojo!), movemos un(os) fichero(s) dentro de un directorio (si estos existen en el *directorio_destino* son reemplazados) y renombramos un directorio (si *nuevo_directorio* existe, será reemplazado). La opción `-i` hace que `mv` nos pida confirmación en el caso de que exista un reemplazamiento. La opción `-` hace que todos los siguientes argumentos

[2]recordar que ◇ significa peligro. Cuidado con este comando porque UNIX no dipone de una utilidad formal para recuperar ficheros borrados con **mv**.

sean tratados como nombres de ficheros, cosa que es útil cuando trabajamos con ficheros que comienzan por el signo - , aunque esta opción no trabaja bien en Linux (por lo menos hasta la construcción de este documento). Ejemplos:

```
gogh:~/doc/Linux> ls
alumnos.txt      mo_pro.txt       seleccion_1.txt
mo_pro.dos       mo_pro2.txt
gogh:~/doc/Linux> mv alumnos.txt Alumnos.txt
gogh:~/doc/Linux> ls
Alumnos.txt      mo_pro.txt       seleccion_1.txt
mo_pro.dos       mo_pro2.txt
gogh:~/doc/Linux> mv -i Alumnos.txt mo_pro2.txt
mv: replace ‘mo_pro2.txt’? n
gogh:~/doc/Linux> ls
Alumnos.txt      mo_pro.txt       seleccion_1.txt
mo_pro.dos       mo_pro2.txt
gogh:~/doc/Linux> cat > -h
hola
gogh:~/doc/Linux> ls
-h               mo_pro.dos       mo_pro2.txt
Alumnos.txt      mo_pro.txt       seleccion_1.txt
gogh:~/doc/Linux> mv - -h 1         (en Linux no podemos usar la opción -)
mv: illegal option -- h
Try ‘mv --help’for more information.
gogh:~/doc/Linux> cd ..             (pero existen triquiñuelas)
gogh:~/doc> mv Linux/-h Linux/h     (renombramos el fichero -h por h)
gogh:~/doc> cd Linux
gogh:~/doc/Linux> ls
Alumnos.txt      mo_pro.dos       mo_pro2.txt
h                mo_pro.txt       seleccion_1.txt
gogh:~/doc/Linux>
```

1.5.10 Creando un directorio.

Podemos crear un nuevo directorio con el comando mkdir . La sintaxis de esta orden es:

mkdir *[-p] {nombre_directorio}+*

La opción -p hace que si el directorio ya existe, este no sea creado (el comando no trabaja). Por supuesto, deberemos de tener permiso de escritura en el directorio del que estamos colgando el nuevo directorio, para poder crearlo. Un ejemplo de su uso podría ser:

```
gogh:~/doc/Linux> ls
Alumnos.txt      mo_pro.txt       seleccion_1.txt
mo_pro.dos       mo_pro2.txt
gogh:~/doc/Linux> mkdir muevo_dir       (creamos un nuevo directorio)
gogh:~/doc/Linux> ls
Alumnos.txt      mo_pro.txt       muevo_dir/
mo_pro.dos       mo_pro2.txt      seleccion_1.txt
gogh:~/doc/Linux> mkdir nuevo_dir/      (intentamos crearlo de nuevo)
mkdir: cannot make directory ‘muevo_dir/’: No such file or directory
gogh:~/doc/Linux> mkdir -p muevo_dir/    (intentamos crearlo sabiendo que puede existir)
gogh:~/doc/Linux>                        (no obtenemos mensaje de error)
```

1.5.11 Copiando un fichero.

Podemos copiar un fichero, con lo que crearemos uno nuevo escribiendo:

- **cp** *-i antiguo_fichero nuevo_fichero*
- **cp** *-ir {fichero}* directorio*

Con la primera forma, duplicamos (físicamente, byte a byte) el contenido de un fichero. Si el fichero nuevo existe, es reemplazado. La opción -i nos preguntará confirmación antes de que esta acción destructiva de datos sea ejecutada. En la segunda versión, aparece la opción -r que ejecutará la orden de copiado de forma recursiva para cada uno de los ficheros y directorios que formen parte del subárbol de fichero. Los permisos de ejecución, lectura y escritura de los nuevos ficheros creados serán idénticos a los de sus originales. Como ejemplos:

```
gogh:~/doc> ls
19_Abril_95.tar.Z  Edinburgh/         curricul/          mate/
Albacete/          Linux/             doctorad/          tesina/
Canarias/          arquitec/          libros/            tesis/
gogh:~/doc> cp -r Linux linux
cd linux
gogh:~/doc> cd linux
gogh:~/doc/linux> ls
Alumnos.txt        mo_pro.txt       seleccion_1.txt
no_pro.dos         mo_pro2.txt
gogh:~/doc/linux>
```

1.5.12 Borrando un directorio vacío.

La orden de borrado de un directorio que no contenga ficheros es rmdir . Su uso es:

- **rmdir** *{directorio}+*

Como puede verse, es posible dar una lista de directorios a borrar, y únicamente borraremos aquellos directorios sobre los que tengamos permisos de escritura. Ejemplos:

```
gogh:~/doc/linux> ls
Alumnos.txt      mo_pro.txt      seleccion_1.txt
mo_pro.dos       mo_pro2.txt
gogh:~/doc/linux> rm *
gogh:~/doc/linux> cd ..
gogh:~/doc> rmdir linux/
gogh:~/doc> ls
19_Abril_95.tar.Z  Edinburgh/         curricul/          mate/
Albacete/          Linux/             doctorad/          tesina/
Canarias/          arquitec/          libros/            tesis/
gogh:~/doc>
```

1.5.13 Borrando ficheros y directorios no vacíos.

La utilidad rm permite eliminar ficheros y directorios del árbol de ficheros y directorios del sistema de ficheros. Cuando borramos un fichero se elimina la entrada con el nombre del fichero y el fichero físicamente salvo en el caso de que estemos referenciando un fichero a través de un enlace simbólico. Los enlaces simbólicos permiten que el mismo fichero tenga más de un nombre distinto (lo mismo puede ocurrir con un directorio). Esto es útil para reducir la cantidad de información físicamente almacenada en el sistema de ficheros. La descripción para rm es:

- **rm** *[-] -fir {fichero o directorio}**

Las opción $\boxed{\text{-i}}$ nos pedirá confirmación antes de cualquier borrado, a lo que deberemos de responder pulsando la tecla $\boxed{\text{y}}$ o $\boxed{\text{n}}$ (y en algunas ocasiones apreciaremos la utilidad del comando $\boxed{\text{yes}}$), con $\boxed{\text{-r}}$ causa que todos los ficheros y directorios que cuelgan del directorio que estamos borrando sean también borrados. Antes de nada, cuidado con este comando y sobre todo cuando somos super usuario y estamos utilizando la opción $\boxed{\text{-r}}$. Como ejemplos:

```
gogh:~/doc/linux> ls
Alumnos.txt      mo_pro.txt      seleccion_1.txt
no_pro.dos       mo_pro2.txt
gogh:~/doc/linux> cd ..
gogh:~/doc> rm -r linux   (borramos un directorio recursivamente)
gogh:~/doc> ls
19_Abril_95.tar.Z  Edinburgh/        curricul/          mate/
Albacete/          Linux/            doctorad/          tesina/
Canarias/          arquitec/         libros/            tesis/
gogh:~/doc> ls /usr/local/bin     (vemos el contenido de otro directorio)
gnuchess*  gnuchessc*  gnuchessn*  gnuchessr*  gnuchessx*  xboard*
gogh:~/doc> ln -s /usr/local/bin chess  (creamos un enlace simbólico)
gogh:~/doc> ls -l
total 358
-rw-rw-r--   1 vi       wheel      338973 Apr 19  1995 19_Abril_95.tar.Z
drwxrwxr--   2 vi       wheel         512 Apr 18  1995 Albacete/
drwxrwxr--   2 vi       wheel        1536 Apr 19  1995 Canarias/
drwxrwxr--   2 vi       wheel         512 Apr  4 16:52 Edinburgh/
drwxrwxr--   2 vi       wheel         512 Apr 19  1995 Linux/
drwxrwxrwx  10 vi       wheel        1024 Feb 15 10:20 arquitec/
lrwxrwxrwx   1 vi       wheel          14 Apr 19  1995 chess -> /usr/local/bin/
drwxrwxrwx   2 vi       wheel         512 Mar 21 18:58 curricul/
drwxrwxrwx   2 vi       wheel         512 May  3  1994 doctorad/
drwxrwxrwx   2 vi       wheel         512 Nov 17 13:22 libros/
drwxrwxr--   3 vi       wheel         512 Feb  6 13:57 mate/
drwxrwxrwx   2 vi       wheel        1536 Apr 14 19:03 tesina/
drwxrwxrwx   2 vi       wheel         512 Oct 17  1994 tesis/
gogh:~/doc> ls chess
gnuchess*  gnuchessc*  gnuchessn*  gnuchessr*  gnuchessx*  xboard*
gogh:~/doc> rm chess   (borramos el enlace simbólico)
gogh:~/doc> ls /usr/local/bin   (pero no el contenido del directorio físico)
gnuchess*   gnuchessc*  gnuchessn*  gnuchessr*  gnuchessx*  xboard*
gogh:~/doc>
```

1.5.14 Listando los grupos a los que pertenecemos.

En ocasiones es interesante averiguar los grupo de trabajo a los que pertenece un usuario, sobre todo cuando encontramos problemas con los permisos a la hora de copiar o mover ficheros. Los grupos de trabajo han sido creados para que personas que trabajan con los mismos programas y ficheros puedan compartirlos. Esto se puede hacer con:

groups *[nombre_de_usuario]*

Ejemplo:

```
gogh:~> groups vi
vi : wheel
gogh:~> groups root
root : root bin daemon sys adm disk wheel shadow
gogh:~>
```

1.5.15 Cambiando ficheros de grupo.

Es posible cambiar de grupo ficheros de los que somos los dueños. El super usuario puede cambiar cualquier fichero de grupo. La sintaxis del comando es:

⋄ **chgrp** *-R identificativo_de_grupo* {*nombre_de_fichero*}*

La opción $\boxed{\text{-R}}$ permite cambiar el grupo de todos los ficheros de un directorio de forma recursiva. En nuestro caso, como usuarios normales, únicamente pertenecemos a un grupo, por lo que es inútil usar este comando. Sin embargo, el super usuario posee más de un grupo:

```
Welcome to Linux 1.0.8.

gogh login: root (entramos como super usuario)
Password:
Last login: Mon Apr 17 04:09:21 on tty4
Linux 1.0.8. (Posix).
gogh:~# pwd
/root
gogh:~# ls -l
total 1
lrwxrwxrwx   1 root     root      8 Oct 20  1994 INSTALL -> /usr/adm/
drwxr-xr-x   2 root     root   1024 Mar 26 02:29 doom/
lrwxrwxrwx   1 root     root     14 Oct 20  1994 linux -> /usr/src/linux
gogh:~ # groups (averiguamos los grupos a los que el root pertenece)
root bin daemon sys adm disk wheel shadow
gogh:~ # chgrp sys doom (cambiamos de grupo el doom)
gogh:~ # ls -l
total 1
lrwxrwxrwx   1 root     root      8 Oct 20  1994 INSTALL -> /usr/adm/
drwxr-xr-x   2 root     sys    1024 Mar 26 02:29 doom/
lrwxrwxrwx   1 root     root     14 Oct 20  1994 linux -> /usr/src/linux
gogh:~#
```

1.5.16 Cambiando el propietario de un fichero.

Si somos super usuario, podemos cambiar un fichero de dueño con el comando:

⋄ **chown** *-R nombre_nuevo_dueño* {*nombre_de_fichero*}+

donde $\boxed{\text{-R}}$ provoca el cambio de dueño a todos los ficheros y directorios de un directorios de forma recursiva. Ejemplo:

```
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 vi       wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~> su (queremos ser super usuario)
Password: (el password del super usuario)
bash# chown root carta (cambiamos de dueño la carta)
bash# exit (dejamos de ser super usuario)
gogh:~> ls -l
total 3
drwxr-xr-x   2 vi       wheel        1024 Feb 14 01:50 bin/
-rw-rw-r--   1 root     wheel         325 Apr  2 18:49 carta
lrwxrwxrwx   1 vi       wheel          21 Feb 14 01:48 doc -> /home1/paralelo/v/doc/
lrwxrwxrwx   1 vi       wheel          19 Feb 14 01:52 f -> /home1/paralelo/v/f/
-rw-rw-r--   1 vi       wheel         227 Mar 15 20:21 un_fichero_ASCII
gogh:~>
```

1.5.17 Cambiando temporalmente de grupo de trabajo.

En el caso de que pertenezcamos a más de un grupo de trabajo, los ficheros que creamos pertenecerán al grupo que en ese momento esté activo. Podemos cambiar de grupo de trabajo con el comando:

newgrp *[-] [nombre_de_grupo]*

Este comando lo que hace es lanzar un nuevo shell con el nuevo nombre de grupo de trabajo por defecto. El antiguo shell queda dormido hasta que decidimos abandonar el shell, momento en el que la acción del comando newgrp queda anulada. Ejemplos:

```
gogh:~# ls -l
total 1
lrwxrwxrwx   1 root     root            8 Oct 20  1994 INSTALL -> /usr/adm/
drwxr-xr-x   2 root     sys          1024 Mar 26 02:29 doom/
lrwxrwxrwx   1 root     root           14 Oct 20  1994 linux -> /usr/src/linux
gogh:~# newgrp sys
gogh:~# mkdir ajo
gogh:~# ls -l
total 2
lrwxrwxrwx   1 root     root            8 Oct 20  1994 INSTALL -> /usr/adm
drwxr-xr-x   2 root     sys          1024 Apr 23 03:01 ajo
drwxr-xr-x   2 root     sys          1024 Mar 26 02:29 doom
lrwxrwxrwx   1 root     root           14 Oct 20  1994 linux -> /usr/src/linux
gogh:~#
```

1.6 Ficheros ejecutables

Vamos a estudiar la estructura de un fichero ejecutable cuando se encuentra almacenado en disco y que ocurre con él cuando se carga en memoria y recibe la atención de la CPU.

1.6.1 Lanzando ficheros ejecutables

Aunque indirectamente ya hemos aprendido a lanzar ficheros ejecutables, esta sección explica que variedad de ejecuciones podemos realizar. Un fichero ejecutable en UNIX es cualquier fichero que tiene permisos de ejecución. Evidentemente será ejecutable por aquellos usuarios que indiquen los permisos. Para lanzar un programa escribiremos su nombre. Si el fichero ejecutable está almacenado en uno de los directorios indicados en la variable de entorno `$PATH` no es necesario que escribamos el camino completo del fichero, empezando desde el directorio raíz o haciendo relativamente a donde nos encontramos ahora. El nombre del un fichero ejecutable puede ser cualquiera. En UNIX no ocurre como en MS-DOS en que los ficheros ejecutable deben tener como extensión `.EXE` o `.COM`.

El continido de un fichero ejecutable puede ser cualquiera, pero ocurre que no tiene sentido almancenar en un fichero ejecutable código que no sea realmente ejecutable. Lo más común es que este tipo de ficheros almacenen código máquina enlazado (si no está enlazado, el código máquina aún no puede ser ejecutado por el sistema opertivo). La otra posibilidad es que almacenen programas de shell que es lo más parecido a los programas de procesamiento por lotes en MS-DOS y que acaban en la extensión `.BAT`.

Cuando invocamos un programa ejecutable (también llamado en ocasiones comando) desde el shell, este busca en los caminos especificados en la variable `$PATH`. Si el fichero no está en uno de esos caminos el shell necesitará el camino completo. Para averiguar donde está un determinado fichero ejecutable podemos usar el comando:

which {*nombre_fichero_ejecutable*}+

que devuelve una lista con los caminos completos. Por ejemplo:

```
gogh:~$ which ls pwd who
/bin/ls
/bin/pwd
/usr/bin/who
gogh:~$
```

1.6.2 El fichero ejecutable binario

Un ejecutable es un programa que puede ejecutarse por la CPU normalmente para realizar algún trabajo útil. Estos ficheros normalmente se generan con la ayuda de un compilador aunque si se desea también se puede trabajar directamente con el ensamblador. Ya usemos uno u otro, el fichero se enlaza mediante el enlazador (linker) con el resto de librerías del sistema operativo antes de convertirse en un verdadero fichero ejecutable. Estos tienen la siguiente estructura:

1. **Cabecera**: contiene los tamaños del resto de áreas del ejecutable.
2. **Código** (text): almacena el código máquina del programa. No hay datos pero si referencia a los mismos.
3. **Datos** (data): aquí se colocan las variables cuyo valor es conocido por el compilador en tiempo de compilación: constantes, arrays estáticos, etc.
4. **Tabla de símbolos**: almacena las referencias a todos los símbolos (los nombres de las variables usadas) que utiliza el programa. Esta información es útil en tiempo de depuración.
5. **Información de reubicación**: contiene información sobre dónde ha buscado el montador de referencias (linker) que no encontró el compilador.

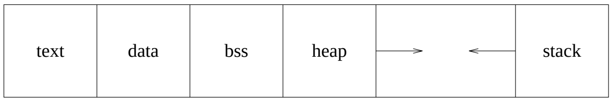

Figura 1.3: Estructura de un proceso.

Las dos últimas secciones no son imprescindibles para el ejecutable si ya está depurado y pueden eliminarse mediante el comando strip . Para averiguar el tamaño de las áreas de código y datos puede usarse el comando size . Ejemplo:

```
gogh:/bin2/bin>  ls -l zrun
-rwxr-xr-x   1 vruiz    wheel       24137 Feb  2 21:12 zrun*
gogh:/bin2/bin> size zrun
text    data    bss     dec     hex     filename
12256   4096    0       16352   3fe0    zrun
gogh:/bin2/bin> strip zrun
gogh:/bin2/bin> ls -l zrun
-rwxr-xr-x   1 vruiz    wheel       16384 Jul  5 19:30 zrun*
gogh:/bin2/bin> size zrun
text    data    bss     dec     hex     filename
12256   4096    0       16352   3fe0    zrun
gogh:/bin2/bin>
```

1.6.3 Imagen de un proceso

Cuando un usuario solicita ejecutar un fichero binario, el sistema operativo carga en una zona de memoria independiente del espacio de direcciones las áreas de código y de datos del ejecutable. A continuación añade un área extra de memoria que contiene la siguiente información (ver figura 1.3):

1. **bss**: es un área de memoria reservada para aquellas variables no han sido inicializadas en tiempo de compilación y si en tiempo de ejecución. Su tamaño se conoce en tiempo de compilación y por eso size puede calcularlo.

2. **heap**: espacio de direcciones de memoria reservada para variables dinámicas, es decir, creadas en tiempo de ejecución y que pueden variar en número y tamaño. Por ejemplo, cuando cuando asignamos memoria a un array dinámico con la instrucción *malloc()* del lenguaje C, reservamos memoria del heap. Normalmente el heap crece desde direcciones decrecientes a direcciones crecientes, aunque esto para el programador es irrelevante.

3. **stack**: es el área de memoria dedicada a gestionar las llamadas y retornos de las funciones invocadas. Típicamente, cuando llamamos recursivamente a una función indefinidamente y se produce un fallo de segmentación (nuestro proceso escribe en una dirección de memoria que no le ha sido asignada) es que hemos llenado la memoria dinámica que fue asignada con el stack. Normalmente el stack crece desde direcciones crecientes a direcciones decrecientes.

Pero el ejecutable cargado en memoria de esta manera no es suficiente. El sistema operativo tiene que multiplexar la CPU entre todos los procesos que se ejecutan en la computadora. Para ello necesita añadir información extra que le permita asignar y retirar la CPU a un proceso. El núcleo del sistema operativo inserta al ejecutable un entorno que contiene esa información, típicamente: contenido de los registros de la CPU, directorio de trabajo, estado de los ficheros abiertos en el instante en el que perdió la atención de la CPU, recursos consumidos, etc. A la combinación de ejecutable antes descrita y el entorno se le denomina **imagen**. Esta estructura si contiene toda la información necesaria para que un proceso se ejecute creyendo poseer todos los recursos del ordenador.

1.6.4 Los procesos en UNIX

Cuando una imagen recibe el control de la CPU se le denomina **proceso**. Sin embargo, como en un intervalo de tiempo suficientemente grande todas las imágenes van a ejecutarse, por abuso del lenguaje, se confunde el concepto de imagen al de proceso. El sistema operativo identifica cada proceso mediante un número entero único denominado **identificador de proceso: PID** (Process IDenfiticator). Un proceso es interactivo cuando necesita de el usuario para ejecutarse convenientemente usando para ello un terminal, de lo contrario se denomina no interactivo o batch.

Evolución de un proceso

Cuando un usuario está en su cuenta, le atiende el intérprete de comandos (shell), que es un proceso con un cierto identificador que puede averiguarse mediante el comando de usuario ps -f (la opción -f genera un árbol con las dependencias entre procesos)[3]. Por ejemplo:

```
gogh:~> ps -f
PID TTY STAT  TIME COMMAND
611 pp0 S     0:01 -csh
695 pp0 R     0:00 ps -f
gogh:~>
```

Cuando el usuario solicita ejecutar un comando, como por ejemplo who para ver cuáles son los otros usuarios conectados al sistema, es necesario que nazca un nuevo proceso que soporte la ejecución de dicho comando. Cuando el shell (proceso padre) recibe la cadena ”who”, ejecuta la llamada al sistema operativo *fork()*, para crear un proceso hijo que es una copia idéntica a él mismo (shell) en lo que se refiere al ejecutable y a una parte del entorno. Obsérvese que el PPID (Parent PID) del proceso hijo es el PID del proceso padre. Así pues, hay dos procesos disjuntos, `shell` y `who`, que ejecutan lo mismo: el intérprete de comandos. A continuación el proceso padre se queda esperando con la llamada al sistema operativo *wait()* hasta que su hijo se ejecute. Mientras tanto el proceso hijo ejecuta la llamada *exec()* para cambiarse a si mismo su código ejecutable, con lo que sustituye el código del intérprete de comandos por el ejecutable binario del comando de usuario who , que corresponde al fichero `/bin/who`. A partir de este momento, el proceso hijo está ejecutando el comando de usuario who , y no el intérprete como había sucedido hasta ahora. Cuando el proceso hijo concluye su ejecución, ejecuta la llamada *exit()* que envía una señal *signal()* al proceso padre, la cual le despierta y permite continuar su ejecución mostrando el símbolo de espera de nuevo comando (normalmente: `>` o `$` en la pantalla del terminal).

Generación de los procesos del sistema

Mediante la descripción dada en el apartado **Evolución de un proceso**, queda explicada la genealogía de procesos en UNIX. Tan solo falta un pequeño detalle: ¿ cómo ha nacido el intérprete de comandos ?. La explicación está en la figura 1.4.

[3] Los flags pueden cambiar con la versión de UINIX. Por ejemplo, bajo Linux la opción -f provoca que se muestre un árbol de procesos y no el listado ampliado.

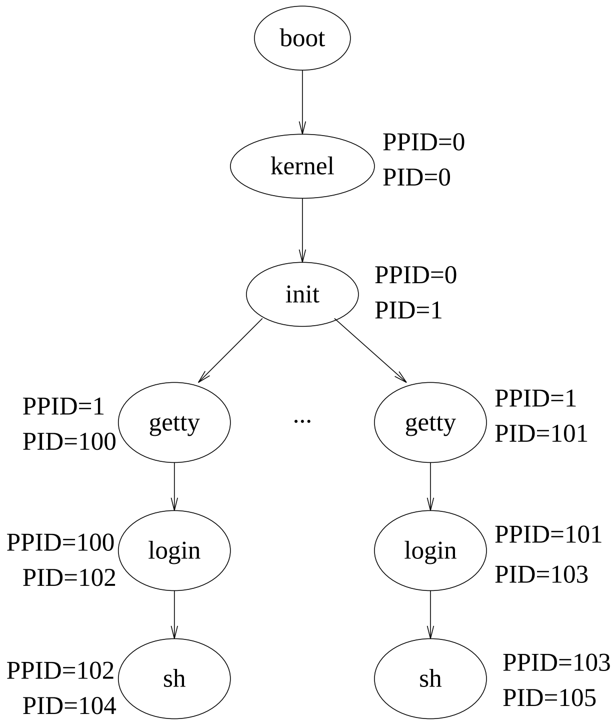

Figura 1.4: Ejemplo sobre la creación de procesos.

Cuando la computadora arranca el sistema operativo, se ejecuta el fichero ejecutable `kernel` cuya misión es cargar el núcleo del sistema operativo en memoria. El núcleo es el primer proceso del sistema, y es padre de si mismo, puesto que su PID=PPID=0. La misión del núcleo es controlar los recursos de la computadora, por lo que delega el nacimiento de nuevos procesos en uno de sus hijos: `init`. Para cada uno de los terminales datos de alta en el ordenador y que es ese instante están usándose (tienen un usuario golpeando sus teclas) `init` crea un proceso hijo en el que se ejecuta el programa `getty` que solicita la conexión. Cuando ésta se obtiene, el mismo proceso se transforma en el programa `login` que solicita el nombre de usuario y la contraseña (si esta fue establecida). Si ambas informaciones son correctas, este proceso se convierte en el intérprete de comandos.

Mediante el comando $\boxed{\text{ps}}$ puede comprobarse la genealogía de los procesos del sistema. Los parámetros cambian dependiendo de la versión de UNIX. Por ejemplo en Linux, para obtener la genealogía de nuestros procesos podemos escribir:

```
bash# ps -xfl
F    UID   PID  PPID PRI NI SIZE  RSS WCHAN     STAT TTY   TIME COMMAND
0     0     1     0   1  0   44   84 1161df     S     ?    0:01 init
0     0     6     1   1  0   32   52 1164d6     S     ?    0:00 (update)
0     0     7     1   1  0   32   80 10ff90     S     ?    0:00 update(bdflush)
0     0    21     1   1  0   60  172 10ff90     S     ?    0:04 /usr/sbin/crond -l10
0     0    37     1   1  0   69  140 130b5d     S     ?    0:01 /usr/sbin/syslogd
0     0    39     1   1  0   44   96 1164d6     S     ?    0:00 /usr/sbin/klogd
0     0    41     1   1  0   72   88 130b5d     S     ?    0:00 /usr/sbin/inetd
0     0   932    41   9  0   46  252 130b5d     S     ?    0:00  \_ in.rlogind
0   999   933   932   1  0  386  540 10ff90     S    pp0   0:01      \_ -csh
0     0   965   933   5  0  381  516 11b8a8     S    pp0   0:00          \_ bash
0     0   987   965  28  0   65  196 0          R    pp0   0:00              \_ ps -xfl
0     0    43     1   1  0   64    0 130b5d     SW    ?    0:00 (lpd)
0     0    50     1   1  0  251  184 1164d6     S     ?    0:01 sendmail: accepting connections
0     0    59     1   1  0   41    0 1a3dd7     SW   v04   0:00 (agetty)
0     0    60     1   1  0   41    0 1a3dd7     SW   v05   0:00 (agetty)
0     0    61     1   1  0   41    0 1a3dd7     SW   v06   0:00 (agetty)
0     0   130     1   1  0   41    0 1a3dd7     SW   v03   0:00 (agetty)
0     0   500     1   1  0   41  108 1a3dd7     S    v02   0:00 /sbin/agetty 38400 tty2
0   999   691     1   1  0 1117 1880 130b5d     S     ?    1:14 xfig
0     0   708     1   1  0   41  220 1a3dd7     S    v01   0:00 /sbin/agetty 38400 tty1
bash#
```

Para conocer el significado de cada columna lo más recomendable es hacer $\boxed{\text{man ps}}$ y leer con tranquildad las especificaciones de ese sistema operativo.

El control de procesos en UNIX

Como sabemos, UNIX es un sistema operativo multitarea, por lo que en todo momento es normal que encontremos más de un proceso ejecutándose y por lo tanto residente en memoria principal. Es esta sección estudiaremos los comandos que permiten gestionan los procesos concurrentes:

1. **El estado de los procesos.**
   El comando que averigua cuales son los procesos que se están ejecutando en un instante determinado ya es conocido: $\boxed{\text{ps}}$ (**P**rocess **S**tatus):

   (a) **ps** *-arux*

   El comando $\boxed{\text{ps}}$ genera una lista con información sobre el estado de los procesos. Por defecto (sin opciones) la lista se limita a los procesos creados en el shell actual. La opción $\boxed{\text{-a}}$ presenta los procesos de todos los usuarios. La opción $\boxed{\text{-x}}$ lista los procesos en background que están corriendo pero que fueron creados en sesiones anteriores. La opción $\boxed{\text{-r}}$ restringe la salida a los procesos que están corriendo. La opción $\boxed{\text{-u}}$ informa extensamente acerca de todos los procesos que hemos seleccionado con el resto de las opciones. La información presentada es la siguiente:

Columna	Significado
USER	Dueño del proceso.
PID	Identificativo del proceso.
%CPU	Porcentaje de CPU que el proceso ha consumido en el último minuto.
%MEM	Porcentaje de memoria RAM que el proceso ha consumido en el último minuto.
SZ	Tamaño en KBytes de la pila y de los datos del proceso.
RSS	Tamaño en Kbytes de memoria RAM que el proceso tiene asignada.
TT o TTY	Terminal de control del proceso (del cual fué lanzado).
STAT	Estado del proceso (una letra de la siguiente tabla).
START	Momento en el que el proceso fué creado.
TIME	Cantidad de CPU consumida en segs.
COMMAND	Nombre del comando que ha creado el proceso.

Dónde el campo `STAT` toma los siguientes valores en función de si el proceso está:

Caracter	Significado
R	Corriendo.
T	Suspendido.
P	Esperando para paginar.
D	En espera ocupada no interrumpible (como esperar una I/O del disco).
S	Durmiendo menos de 20 segs.
I	Durmiendo más de 20 segs.
Z	Zombie.

En el caso de que `STAT` tenga una de las letras anteriores seguida de la letra `N` significa que la prioridad actual del proceso ha sido modificada con el comando nice . Si sigue una `W`, significa que el proceso está "swapped out". Esto quiere decir que el proceso se está ejecutando, pero sus páginas de código se han escrito en el disco para dejar espacio en la memoria de la computadora para que otro proceso se puede ejecutar. Valga como ejemplos:

```
gogh:~> ps (preguntamos por los procesos que ejecuto concurrentemente)
  PID TTY STAT  TIME COMMAND
   58 v01 S     0:01 -csh (un C Shell)
   59 v02 S     0:00 -csh
   71 v02 S     0:01 telnet iron (una conexión a otra estación llamada íron')
   72 v01 S     1:12 vi shells.tex i (el documento que estoy escribiendo)
  160 v03 S     0:01 -csh
  191 v03 R     0:00 ps (el programa que muestra los procesos)
gogh:~> ps -u (deseo toda la información acerca de mis procesos)
USER       PID vi         58  0.0  5.6  346  156 v01 S    15:53   0:01 -csh
vi          59  0.0  6.6  346  184 v02 S    15:53   0:00 -csh
vi          71  0.0  1.8  112   52 v02 S    15:56   0:01 telnet iron
vi          72  0.6  7.7  168  216 v01 S    15:58   1:15 vi shells.tex
vi         160  0.0 11.9  348  332 v03 S    18:34   0:01 -csh
vi         192  0.0  8.0   72  224 v03 R    19:12   0:00 ps -u
gogh:~>ps -a (información acerca de todos los procesos)
  PID TTY STAT  TIME COMMAND
   61 v04 S     0:00 -bash
   62 v05 SW    0:00 (agetty)
   63 v06 SW    0:00 (agetty)
  174 v04 S     0:00 -sh
   58 v01 S     0:01 -csh
   59 v02 S     0:00 -csh
   71 v02 S     0:01 telnet iron
   72 v01 S     1:16 vi shells.tex
  160 v03 S     0:01 -csh
  193 v03 R     0:00 ps -a
gogh:/> find . -name carta -print > $HOME/busqueda & (en background)
[1] 203 (número que identifica el proceso en background)
gogh:/> ps -r (procesos que están corriendo)
  PID TTY STAT  TIME COMMAND
  203 v03 R     0:06 find . -name carta -print
  205 v03 R     0:00 ps -r
gogh:~>
[1]    Done                  find . -name carta -print
gogh:~> (sleep 100; echo He dormido durante 100 segs) &
[1] 285
gogh:~> ps -x (preguntamos por los procesos corriendo en background)
  PID TTY STAT  TIME COMMAND
   58 v01 SW    0:01 (csh)
   59 v02 SW    0:00 (csh)
   71 v02 S     0:03 telnet iron
   72 v01 S     1:42 vi shells.tex
  279 v03 S     0:01 -csh
  285 v03 S     0:00 -csh
  286 v03 S     0:00 sleep 100 (este proceso duerme en background 100 segs)
  287 v03 R     0:00 ps -x
gogh:~>
gogh:~> He dormido durante 100 segs
[1]    Done                   ( sleep 100; echo He dormido durante 100 segs )
gogh:~>
```

2. **Matando procesos.**
   En el caso de que deseemos que un proceso finalice su ejecución antes de que acabe por su normal realización, podemos usar el comando:

   ◇ **kill** *[-identificativo_de_señal]* {*número_de_proceso*}+
   kill *-l*

   Normalmente, no se hace necesario recurrir a una señal poderosa para acabar con la ejecución de la mayoría de los procesos, pero cuando alguno se resiste, la opción `-9` (muerte incondicional) hace maravillas. La opción `-l` provoca la obtención de una lista con todos los posibles identificativos de señales que podemos enviar a los procesos. Ejemplos:

```
gogh:~> sleep 1000 & (ejecutamos en background un proceso que espera 1000 segs)
[1] 29
1 gogh:~> ps -r (procesos ejecutándose)
  PID TTY STAT  TIME COMMAND
  292 v03 R     0:00 ps -r (este es el único que consume CPU)
gogh:~> ps -x (procesos ejecutándose y en background)
  PID TTY STAT  TIME COMMAND
   58 v01 SW    0:01 (csh)
   59 v02 SW    0:00 (csh)
   71 v02 S     0:03 telnet iron
   72 v01 S     1:48 vi shells.tex
  279 v03 S     0:01 -csh
  291 v03 S     0:00 sleep 1000 (el proceso de espera esta dormido)
  293 v03 R     0:00 ps -x
gogh:~> kill 291 (matamos el proceso de espera)
[1]    Terminated             sleep 1000
gogh:~> ps -x (efectivamente el proceso ha sido eliminado)
  PID TTY STAT  TIME COMMAND
   58 v01 SW    0:01 (csh)
   59 v02 SW    0:00 (csh)
   71 v02 S     0:03 telnet iron
   72 v01 S     1:50 vi shells.tex
  279 v03 S     0:01 -csh
  294 v03 R     0:00 ps -x
gogh:~>
```

Existen procesos que no se pueden matar con kill porque cambian su PID dinámicamente. El proceso de matanza es en esto casos especial y más laborioso.

1.6.5 Usando del *background*

En UNIX podemos invocar los comandos de dos formas diferentes: (1) en *foreground* y (2) en *background*. La primera forma es la que hemos usado hasta ahora en casi todas las ocasiones y se utiliza para lanzar comandos interactivos desde el shell. Estos comandos se caracterizan porque se ejecutan en poco tiempo y por tanto devuelven el control al shell (que es el proceso que los ejecuta) rápidamente. Pero puede ocurrir que ejecutemos procesos que tardan bastante tiempo en ejecutarse (en teoría no hay límite: podemos hablar de meses) y no es agradable que el shell se quede bloqueado hasta que el proceso acabe, más aún cuando no es posible trabajar en modo multisesión. Así, podemos usar la ejecución en *background* que proporciona el sistema operativo UNIX para lanzar ese proceso pesado en paralelo con el resto de procesos que corren en el sistema, entre los que evidentemente se encuentra el shell. Para lanzar un proceso en *background* sólo tenemos que acabar el comando interactivo con el metacarácter `&`. El siguiente ejemplo envía un mail a un usuario llamado `vruiz` a la computadora `obelix.cica.es`. Como sabemos que el mensaje que vamos a enviarle es muy largo, podemos hacerlo en *background* y dejar que se envié si que nos afecte prácticamente. Esto se hace así:

Cuando el sistema operativo termina de ejecutar el comando, este nos avisa con el mensaje:

Lanzar procesos en *background* se complica un poco cuando estos sacan mensajes en el terminal usando la salida estándar o la salida de error estándar. Cuando esto ocurre, el proceso que corre en *background* ensucia el terminal, lo que no es agradable, aunque este es el único problema que presenta. De todas formas, muchas veces es interesante que estas salidas puedan ser consultadas cuando el programa ha finalizado, cosa que puede ocurrir dentro de un mes. En estos casos, las salidas pueden redirigirse a un fichero o a un dispositivo. EL siguiente ejemplo muestra como se utilizan la salida estándar y la salida de error estándar desde un programa escrito en lenguaje C. Este se compila y se muestra como se redireccionan cada una de las salidas.

Para compilar este programa sólo tenemos que invocar al compilador de C con el nombre del fichero que deseamos compilar:

```
gogh:~$ cc salidas.c
gogh:~$
```

El fichero ejecutable, ya que no hemos indicado lo contrario al compilador se llama `a.out` y se ha creado en el directorio actual.

```
gogh:~$ ls -l a.out
-rwxr-xr-x   1 vruiz    wheel        4159 Feb  6 13:32 a.out
gogh:~$
```

Para ejecutarlo lo invocamos sencillamente si es que el camino `.` está contemplado en nuestra variable de entorno `$PATH`. Supongamos que esto es así:

```
gogh:~$ a.out
he usado la salida estándar
he usado la salida de error estándar
gogh:~$
```

Ha ocurrido que ambas salidas aparecen en el terminal. Para redireccionarlas usaremos los metaracaracteres de redirección que proporciona el shell. Estos usan los *pipes* el UNIX. Si deseamos redireccionar la salida estándar a un fichero escribiremos:

```
gogh:~$ a.out > 1
he usado la salida de error estándar
gogh:~$ cat 1
he usado la salida estándar
gogh:~$
```

Y si lo que deseamos es redirigir s'lo la salida de error estándar:

```
gogh:~$ a.out 2> 1
he usado la salida estándar
gogh:~$ cat 1
he usado la salida de error estándar
gogh:~$
```

Podemos redirigir ambas usando los dos redirectores simultaneamente:

```
gogh:~$ a.out 2> 2 > 1
gogh:~$ cat 1 2
he usado la salida estándar
he usado la salida de error estándar
gogh:~$
```

Por último, y lo que era nuesto objetivo, podemos lanzar el comando en *background*, sin peligro de que el terminal se llene con los mensajes de los procesos que corren en este modo:

Correr los procesos en *background* facilita la ejecución de trabajos pesados, pero que ocurre si necesitamos avandonar el terminal y nuestro proceso aún no ha acabado.

Si hacemos el `logout` en estas condiciones los procesos en *background* mueren porque muere el proceso padre que los lanzó (el shell normalmente). Esto no es deseable y más cuando el proceso ya lleva consumido muchas horas de CPU. Para resolver este problema el sistema operativo dispone del comando `nohup` que permite evitar este inconveniente. Su sintaxis:

nohup {*nombre_fichero_ejecutable*}+ {*args*}*

`nohup` ejecuta un comando haciéndolo inmune a las señales de finalización que pueda enviarse otro proceso. Además redirecciona la salida a un fichero llamado `nohup.out` e incrementa la prioridad del proceso en 5. Las salidas estándar y de error estándar se añaden a este fichero. Si este no puede ser abierto para escritura, se intenta crear en `$HOMEnohup.out`, y si esto falla, `nohup` falla (esto puede ocurrir porque sobre ninguno de los directorios tengamos permiso de escritura). El fichero `nohup.out` se crea sin permisos para el resto de usuarios (`-rw-------`), pero si ya existía los permisos se respetan. Por último, decir que `nohup` no pone automáticamente el comando en *background* sino que hay que hacerlo explícitamente como ya se ha indicado. Un ejemplo:

1.7 La edición de ficheros ASCII.

1.7.1 Editando un fichero (vi).

Aunque parezca lo contrario, el editor de ficheros vi es probablemente uno de los editores de ficheros ASCII más potentes y a primera vista, de los menos amigables. Quizás la culpa de esto sea consecuencia de su principal ventaja: la ínfima cantidad de ancho de banda que consume de la red. Esto hace a vi idóneo para la edición de ficheros en redes lentas. Fue inicialmente desarrollado para el BSD UNIX en Berkeley y posteriormente introducido como una utilidad estándar para el System V. Esta es la razón por la que vi está disponible en todos las versiones del sistema operativo UNIX. Podemos invocar al vi sin o con parámetros. Si lo hacemos sin parámetros obtendremos una pantalla casi vacía en la que únicamente existe un carácter ~ por cada línea vacía que existe en el fichero que estamos creando. Al final, vi nos da el mensaje:

```
"[NO FILE]"  1 line, 1 char
```

que indica que no hemos dado nombre de fichero y que estamos en la primera línea, primer carácter. Supongamos que tenemos un fichero ya creado llamado `fichero.txt`. Para editarlo podemos teclear: vi fichero.txt y el fichero aparecerá en pantalla. Es importante señalar que dependiendo del tipo de terminal que tengamos definido en la variable de entorno `$TERM`, vi actuará consecuentemente. En Linux el tipo de terminal indicado es `con80x25` o `Linux` con lo que vi presenta ciertas cualidades extra. Por ahora hablaremos únicamente de las características básicas y comunes a todas las implementaciones del vi . La modificación de la variable de entorno `$TERM` se estudia en próximas secciones.

Una vez que hemos ejecutado el editor vi , éste espera a que le demos una o varias órdenes. vi tiene dos modos de funcionamiento llamados modo de entrada de texto y modo de comandos. El primero sirve para alterar el texto del fichero que estamos editando y el segundo para movernos por el texto, buscar o reemplazar cadenas, etc.

1. **Introduciendo texto.** Estando en el modo de comandos, para crear o modificar el contenido del fichero que estamos editando pulsaremos una de las siguiente

teclas:

Tecla	Acción
i	Insertar texto antes del cursor.
a	Insertar texto después del cursor.
o	Insertar texto al final de la línea actual.
O	Insertar texto al comienzo de la línea actual.
R	Reemplazar texto a partir de donde está el cursor.

Todo el texto que entremos será presentado en la pantalla y si hemos de borrar caracteres, podemos hacerlo con la tecla backspace . Normalmente, el uso de los cursores para movernos por el texto, no está permitido en el modo de inserción, debido a que es interpretado como caracteres ASCII regulares (cosa que no ocurre en la versión mejorada el vi en Linux). En la versión estándar del vi , deberemos de colocarnos primero en el modo de comandos para usar los cursores. Esto es consigue pulsando la tecla escape .

2. **El modo de comandos.** Para activar este modo deberemos de pulsar la tecla escape . Si estando en modo de comandos pulsamos esta tecla, escucharemos un pitido indicando que ya estamos en dicho modo, pero no ocurre nada. Este es el modo en el realizamos los siguientes tratamientos:

   (a) **Movimiento del cursor.** Estando en el modo de comandos podemos realizar los siguientes movimientos:

Tecla	Acción
k	Subir una línea.
j	Bajar una línea.
l	Un carácter a la derecha.
h	Un carácter a la izquierda.
˜	Ir al comienzo de la línea actual.
$	Ir al final de la línea actual.
b	Retroceder una palabra.
w	Avanzar una palabra.
ˆd	Avanzar media pantalla.
ˆf	Avanzar una pantalla.
ˆu	Retroceder media pantalla.
ˆb	Retroceder una pantalla.
: *nn* enter	Ir a la línea *nn*.

   (b) **Borrando texto.** Partiendo el modo de comandos, podemos utilizar las siguientes teclas para eliminar texto de un fichero ya creado:

Tecla	Acción
x	Borrar el carácter que hay debajo del cursor.
dw	Posiciona el cursor al inicio de la palabra y la borra.
dd	Borra la línea sobre la que está el cursor.
D	Borra a partir del cursor el resto de la línea.
: *rango* d enter	Borra un bloque de líneas.

   Como ejemplos de indicación de rangos:

Entrada	Acción
:3d enter	Borra la línea número tres.
:1,3d enter	Borra las líneas uno, dos y tres.
:1,$d enter	Borra todas las líneas del fichero.
:.d enter	Borra la línea actual.
:.,$d enter	Borra todas las líneas desde la posición actual.
:.-2d enter	Borra la línea situada dos líneas por encima.
:.,.+2d enter	Borra la línea actual y las dos siguientes.

(c) **Reemplazando texto.** Podemos entrar en el modo de sustitución de texto de las siguientes formas:

Entrada	Acción
r	Sustituiremos a partir del carácter que hay debajo del cursor.
cw	Posiciona el cursor al principio de la palabra y reemplaza.
cc	Posiciona el cursor al principio de la línea y reemplaza.

(d) **Pegando texto.** El editor vi posee un buffer de memoria dedicado al copiado y movimientos entre áreas del fichero. Dentro del modo de comandos lo usamos mediante:

Entrada	Acción
: *rango* y enter	Copia el rango indicado en el buffer.
:pu enter	Copia el buffer tras la línea actual.
: *nn* pu enter	Copia el buffer tras la línea nn.

(e) **Insertando texto.** Es posible insertar el contenido de un fichero externo al que estamos editando haciendo :r *nombre_fichero* enter ,que copia el fichero indicado después de la línea actual.

(f) **Buscando texto.** vi permite buscar hacia delante o hacia detrás una cadena de caracteres a partir de la línea actual en la que nos encontramos. Las acciones posibles son:

Entrada	Acción
/ *sss* enter	Busca la cadena *sss* hacia delante.
? *sss* enter	Busca la cadena *sss* hacia detrás.
n	Repite la última búsqueda.

(g) **Buscando y reemplazando texto.** Podemos realizar búsquedas y reemplazamientos globales de cadenas con los siguientes comandos:

Entrada	Acción
: *rango* s/ *sss* / *ttt* / enter	Reemplaza la primera ocurrencia de *sss* por *ttt*.
: *rango* s/ *sss* / *ttt* /g enter	Reemplaza cada ocurrencia de *sss* por *ttt* .

(h) **Salvando y leyendo ficheros.** Una vez que hemos modificado el texto y deseamos salir o modificar otro ya existente, deberemos de usar los comandos:

Entrada	Acción
:w *nombre_fichero* enter	Salva el contenido del fichero editado con el nombre indicado.
:w enter	Salva en contenido con el nombre que lo creamos.
:e *nombre_fichero* enter	Edita un nuevo fichero.
:n enter	Edita el siguiente fichero indicado en la línea de comandos de llamada al vi .
:r *nombre_fichero* enter	Inserta un fichero en la línea actual.

(i) **Otros comandos de interés.** Algunos comandos usados frecuentemente en el editor vi son:

Entrada	Acción
ˆL	Redibuja la pantalla (cosa que se puede realizar también en el modo de edición).
:! *comando* enter	Ejecuta un comando en un subshell.
:q enter	Salimos del editor si ya hemos salvado el fichero.
:q! enter	Salimos del editor sin salvar el fichero editado.
:wq enter	Salimos del editor salvando el fichero.
:x enter	Idéntico al anterior comando.
:set number enter	Enumera ”virtualmente”las líneas del fichero.

Para ejecutarlos debemos estar en el modo de comandos.

1.8 Imprimiendo documentos.

Para imprimir un fichero sobre papel usaremos el comando lpr . Más exactamente:

lpr *-m [-P impresora] [-número_de_copias]* {*fichero*}*

imprime un fichero (ASCII normalmente) utilizando la impresora especificada. En su defecto se usará la impresora definida en la variable de entorno `$PRINTER`. Si no indicamos nombre de fichero, la entrada estándar será imprimida. La opción -m causará que recibamos un mail indicando que la impresión ha finalizado. Si la impresora está ocupada, nuestro trabajo de impresión será almacenado en la cola de impresión y será atendido dependiendo del número que halla sido asignado. Para ver como está la cola de impresión usaremos el comando:

lpq *-l [-P impresora]* {*trabajo*}* {*usuario*}*

que nos enseña el estado de cada trabajo de la cola de impresión asignada a la impresora `-P` <*impresora*>. En su defecto la impresora declarada en la variable de entorno `$PRINTER` será usada. Si no especificamos trabajos de impresión, el estado de cada uno de los trabajos será presentado, y por último, con la opción -l observaremos información extra. En ocasiones, nos arrepentimos de haber enviado un trabajo hacia la impresora, y nuestros deseos pueden hacerse realidad usando el comando lprm . Su sintaxis:

lprm *[-P impresora] [-]* {*trabajo*}* {*usuario*}*

Los únicos trabajos que podemos abortar son los que nos pertenecen, a no ser que seamos super usuario. Como ejemplos de gestión de la cola de impresión tenemos:

```
bash$ ls
Alumnos.txt      mo_pro.txt       seleccion_1.txt
no_pro.dos       mo_pro2.txt
bash$ lpq (vemos la cola de impresión por defecto)
no entries
bash$ lpr -pdj0 Alumnos.txt (imprimimos sobre la impresora dj0)
bash$ lpq (comprobamos la impresora por defecto)
no entries
bash$ lpq -Pdj0 (comprobamos la impresora dj0)
Wed Apr 19 17:36:51 1995: waiting for dj0 to become ready (offline ?)
Rank   Pri Owner      Job  Files                   Total Size
1st    0   vi         104  Alumnos.txt                   1535 bytes
bash$ lprm -Pdj0 1 (borramos el trabajo de la cola (impresora apagada))
bash$ lpq -Pdj0 (el trabajo ha sido borrado)
no entries
bash$
```

Capítulo 2

Los Shells de UNIX

2.1 Introducción

Un **shell** es un programa que corre entre el núcleo del sistema operativo y el usuario para ayudar a controlar todo el sistema. Es el encargado de recoger todas nuestras peticiones, procesarlas y lanzar los programas necesarios. Es directamente responsable de las cualidades de multitarea del UNIX. Haciendo una similitud con el MS-DOS, los shells equivalen al intérprete de comandos `COMMAND.COM`. Como ya indicamos en la sección llamada *Lo Básico de UNIX*, en las diferentes versiones de UNIX, normalmente nos encontramos con los shells sh , csh y ksh . El uso de uno u otro depende del usuario. sh es sencillo y básico, ksh es el más amplio y csh es el más adecuado para el trabajo interactivo. Posteriormente han aparecido shells mejorados entre los que podemos destacar el bash (Bourne-Again SHell) que es una ampliación del sh con ciertas facilidades de uso añadidas o tcsh que es una mejora de csh. En este apartado estudiaremos dos shells diferentes:

1. El Bourne Shell.
2. EL C Shell.

Todos los shells tienen mucho en común, por lo que no son demasiadas las diferencias. Una vez que conocemos uno de ellos, fácilmente migramos hacia otro. Se ha elegido estos dos porque son los más básicos. Conociendo el sh no tendremos problemas cuando migremos hacia ksh o bash y conociendo csh conocemos también tcsh . Existe además un shell llamado zsh que está diponible en la mayoría de las distribuciones de Linux. Es básicamente una ampliación de ksh .

2.2 Lo común a todos los shells

2.2.1 Presentando información

El control de los programas Shell pasa por el estado de sus variables de control. Estas definen todos los parámetros de funcionamiento. Es interesante que antes de nada sepamos como presentar información acerca de las variables que están activadas en en shell que estamos ejecutando. Estas variables (como a continuación veremos) contienen información vital para el shell. El comando que usamos para ello es:

echo *-n* {*arg*}*

Su funcionamiento es similar al que tiene en el sistema operativo MS-DOS y compatibles. Ejemplos de su uso:

```
Welcome to Linux 1.0.8.

gogh login: vi (entramos en Linux)
Password:
Last login: Sun Apr 23 23:06:46 on tty2
Linux 1.0.8. (Posix).
gogh:~> echo Un ejemplo de la utilización del comando echo
Un ejemplo de la utilización del comando echo
gogh:~> echo $TERM (vemos en contenido de la variable TERM)
con80x25
gogh:~> echo La fecha y hora en el sistema es: ‘date‘
La fecha y hora en el sistema es: Sun Apr 23 23:09:53 GMT 1995
gogh:~>
```

La opción `-n` se utiliza si no deseamos que si produzca un salto de línea tras hacer el eco.

2.2.2 Seleccionando un Shell

En el momento en el que el administrador del sistema nos crea una cuenta en el sistema, se nos asigna un shell por defecto. La primera vez que accedemos a la cuenta, podemos averiguar cual es el nuestro, observando el prompt. Otra forma de hacerlo es averiguar el contenido de una variable de entorno llamada `$SHELL`. Esto se pude hacer de la siguiente forma:

```
gogh:~> echo $SHELL
/bin/csh
gogh:~>
```

Esta variable se carga con el contenido mostrado cuando se hacemos el `login`. Si no estamos de acuerdo con la decisión tomada por el system manager (root, superusuario ... es curioso la cantidad de nombres que la gente da a este tipo de cargos dentro de un sistema informático), podemos cambiar el shell de dos formas diferentes:

1. Lanzando un shell nuevo sobre el que ya tenemos lanzado. Esta elección es rápida, pero tiene el problema de que cuando abandonamos la sesión y volvemos a entrar de nuevo, se lanza el shell por defecto.

2. Mediante el comando **chsh** . Aquí tenemos un ejemplo de su funcionamiento:

```
gogh:~> chsh (cambiamos de shell por defecto)
Changing the login shell for vi
Enter the new value, or press return for the default
Login Shell [/bin/csh]: /bin/ksh (elegimos Korn Shell)
gogh:~> exit
logout


Welcome to Linux 1.0.8.
gogh login: vi
Password:
Last login: Sun Apr 23 23:18:01 on tty3
Linux 1.0.8. (Posix).
interactive stdin
<vi@gogh:9>$ (efectivamente hacemos el login con el Korn Shell)
```

2.2.3 Los metacaracteres

En todos los shells existe una serie de caracteres que nos sirven para construir comandos y scripts [1]. En la siguiente tabla se describen:

<table>
<tr><th>Símbolo</th><th>Significado</th></tr>
<tr><td>></td><td>Redirección de salida: escribe la salida estándar sobre un fichero.</td></tr>
<tr><td>>></td><td>Redirección de salida: añade la salida estándar a un fichero.</td></tr>
<tr><td><</td><td>Redirección de entrada: lee la entrada estándar de un fichero.</td></tr>
<tr><td><<toc</td><td>Redirección de entrada: lee la entrada estándar de un script hasta encontrar la palabra toc.</td></tr>
<tr><td>*</td><td>Comodín de sustitución: sustituye a 0 o más caracteres.</td></tr>
<tr><td>?</td><td>Comodín de sustitución: sustituye a 1 caracter.</td></tr>
<tr><td>[...]</td><td>Comodín de sustitución: sustituye cualquier carácter entre corchetes.</td></tr>
<tr><td>`<i>comando</i>`</td><td>Sustitución de comando (ver echo).</td></tr>
<tr><td>|</td><td>Encauzamiento: envía la salida estándar sobre la entrada estándar de otro.</td></tr>
<tr><td>;</td><td>Usado en el secuenciamiento de comandos.</td></tr>
<tr><td>||</td><td>Ejecución condiciónal: ejecuta un comando si el previo ha fallado.</td></tr>
<tr><td>&&</td><td>Ejecución condiciónal: ejecuta un comando si el previo se ha ejecutado.</td></tr>
<tr><td>(...)</td><td>Agrupa comandos.</td></tr>
<tr><td>&</td><td>Ejecuta un comando en background.</td></tr>
<tr><td>#</td><td>Todos los caracteres siguientes son tomandos como comentarios.</td></tr>
<tr><td>$</td><td>Leemos el contenido de una variable.</td></tr>
<tr><td>\</td><td>Extensión de comandos que no caben en una sola línea.</td></tr>
<tr><td>''</td><td>Posibilita el eco de metacaracteres.</td></tr>
</table>

A continuación mostramos ejemplos de cada uno de los metacaracteres:

1. Redirigiendo la salida estándar sobre un fichero texto:

```
gogh:~> ls
bin/            doc@            un_fichero_ASCII
carta           f@
gogh:~> ls > 1 (redirigimos la salida estándar sobre el fichero 1)
gogh:~> cat 1
1
bin/
carta
doc@
f@
un_fichero_ASCII
gogh:~>
```

2. Añadiendo la salida estándar a un fichero texto ya creado:

```
gogh:~> echo eso es todo >> 1 (añadimos una nueva línea al fichero 1)
gogh:~> cat 1
1
bin/
carta
doc@
f@
un_fichero_ASCII
eso es todo
gogh:~>
```

[1] Algo parecido al fichero AUTOEXEC.BAT en MS-DOS.

3. Redirigiendo la entrada estándar desde un fichero ya creado hacia un comando que la usa:

```
gogh:~> sort < 1 (redirigimos la entrada para ordenar el fichero 1)
1
bin/
carta
doc@
eso es todo
f@
un_fichero_ASCII
gogh:~>
```

4. Usando el metacarácter de redirección de la salida de un script sobre un comando:

```
gogh:~> cat birth-day (script que envía una felicitación de cumpleaños)
# Esto es una línea con comentarios
# $1 es un argumento del script
# FinDelScript es una etiqueta que marca la parte del script
# que va a ser redireccionada sobre la utilidad de mail
mail $1 << FinDelScript (redirigimos la salida del script sobre el mail)
Muy buenas $1, (nombre del usuario que felicitamos)

! ! ! F E L I Z  C U M P L E A Ñ O S  ! ! !
Hasta pronto.
- $USER (usuario que envía la felicitación)
FinDelScript (palabra que marca el final del script que es redirigido)
echo Felicitación de cumpleaños enviada a $1
gogh:~> birth-day vi (nos enviamos a nosotros una felicitación)
Felicitación de cumpleaños enviada a vi
gogh:~> mail (comprobamos si tenemos correo)
Mail version 5.3 2/18/88.  Type ? for help.
"/var/spool/mail/vi": 1 message 1 new (vemos que si, nos hemos autoescrito)
>N  1 vi                    Mon Apr 24 00:29  21/352
& (pulsamos la tecla [return] )
Message 1: (vemos nuestra felicitación)
From vi Mon Apr 24 00:29:23 1995
To: vi
Muy buenas vi,
! ! ! F E L I Z  C U M P L E A Ñ O S  ! ! !
Hasta pronto.
- vi

& quit (salimos de la utilidad de correo)
Held 1 message in /var/spool/mail/vi
gogh:~>
```

5. Usando los *wildcards* para sustituir nombres de ficheros:

```
gogh:~/pru> ls -FR (recursivamente listamos el directorio actual)
a.c    b.c    cc.c  dir1/  dir2/
dir1:
d.c  e.e
dir2:
f.d  g.c
gogh:~/pru> ls *.c (cualquier número de caracteres seguidos por ".c")
a.c  b.c  cc.c
gogh:~/pru> ls ?.c (un solo carácter seguido de ".c")
a.c  b.c
gogh:~/pru> ls [ac]* (cualquiera que comience por "a" o "c")
a.c  cc.c
gogh:~/pru> ls [A-Za-z]* (aquellos que comiencen por letras)
a.c  b.c  cc.c
dir1:
d.c  e.e
dir2:
f.d  g.c
gogh:~/pru> ls dir*/*.c (de los directorios que empiezan por "dir", los que acaban en ".c")
dir1/d.c  dir2/g.c
gogh:~/pru> ls */*.c (de cualquier directorio, los que acaban ebn ".c")
dir1/d.c  dir2/g.c
gogh:~/pru> ls *2/?.? ?.? (de los directorios acabados en "2", ficheros con dos caracteres)
a.c      b.c      dir2/f.d  dir2/g.c (separados por un punto, ídem en el directorio actual)
gogh:~/pru>
```

6. Como ejemplo de la sustitución de comandos ver el apartado del comando echo . Este ejemplo nos sirve también para comprobar el uso del símbolo `$`.

7. Encauzando las salidas y las entradas estándar: el *pipeline* (o uso de *pipes*) es una filosofía muy utilizada en UNIX, que consiste en dividir grandes problemas en subtareas que pueden ser resueltas por pequeños procesos, que a su vez pueden ser reutilizados en la resolución de otras tareas.

```
gogh:~> ls
1                 birth-day*        doc@              pru/
bin/              carta             f@                un_fichero_ASCII
gogh:~> ls | wc -w (contando el número de entradas en el directorio)
8
gogh:~> cat /etc/passwd | awk -F: '{print $1}'| sort
adm
bin (lista ordenada de los usuarios del sistema)
daemon
ftp
games
halt
lp
mail
man
news
operator
postmaster
root
shutdown
sync
uucp
vi
gogh:~>
```

8. Separando comandos mediante el carácter ; :

```
gogh:~> ls;date
1                 birth-day*        doc@              pru/
bin/              carta             f@                un_fichero_ASCII
Mon Apr 24 01:24:10 GMT 1995
gogh:~>
```

9. Ejecución condicional: todo proceso en UNIX termina con un valor de salida. Por convención, una finalización con éxito representa un retorno del valor 0 y un error devuelve un valor distinto de 0 (normalmente 1, esto puede ser programado cuando se construye el programa ejecutable):

```
iron% cat hola.c
#include <stdio.h>
main() {
        printf("Hola !\n'');
}
iron% cc hola.c && a.out (compilamos y ejecutamos si hemos compilado con éxito.)
Hola !
iron% cc Hola.c || echo Ese programa no existe
cpp: Hola.c: No such file or directory
Ese programa no existe
iron%
```

10. Agrupando comandos:

```
gogh:~> (pwd; date; ls) > 1
gogh:~> cat 1
/usr/users/v (pwd)
Mon Apr 24 02:03:28 GMT 1995 (date)
1 (ls)
bin/
birth-day*
carta
doc@
f@
pru/
un_fichero_ASCII
gogh:~>
```

11. Utilizando el **background**: esta es la forma más primitiva y a la vez actual de utilizar la multitarea en UNIX. Es muy útil, pues permite realizar dos o más tareas al mismo tiempo (en pseudo-concurrencia si sólo disponemos de un procesador o en concurrencia real si se dipone de más de una CPU):

```
gogh:~> find . -name a.c -print | mail vi & (buscamos el fichero a.c)
[1] 84 85
gogh:~> cc hola.c (y mientras compilamos el fichero hola.c)
[1]    Done                  find . -name a.c -print | mail vi
gogh:~> mail (leo mi mail)
Mail version 5.3 2/18/88.  Type ? for help.
"/var/spool/mail/vi": 2 messages 1 new
.   1 vi                    Mon Apr 24 00:29  22/363 (un mail antiguo)
>N  2 vi                    Mon Apr 24 16:06  14/264 (y el nuevo)
& (pulsamos return o 2 )
Message 2:
From vi Mon Apr 24 16:06:05 1995
To: vi
./pru/a.c (localización del fichero a.c)
& q (salimos de la utilidad de mail)
Held 2 messages in /var/spool/mail/vi
gogh:~>
```

Este ejemplo puede no ser demasiado claro en en sentido de que es necesario que el comando `find` debe finalizar no más tarde que el comando `mail`, pero tiene que tenerse en cuenta que ambos comandos se ejecutaron simultaneamente. Esta característica si se aprecia cuando necesitamos realizar dos tareas diferentes sobre un volumen de datos muy grande y elazamos ambas tareas mediante *pipes*. Notaremos que en cuanto la primera tarea ha generado datos de salida la tarea que espera comienza a actuar sobre estos datos, mucho antes de que todo el volumen de datos haya atravesado la primera tarea. Esto ayuda a optimizar el uso de los recursos del sistema, entre ellos el más importante: la CPU.

12. Escribir comandos largos es tan fácil como esto:

```
gogh:~> echo hola \
? caracola
hola caracola
gogh:~>
```

13. Y por último, podemos trabajar con los metacaracteres:

```
gogh:~> echo también es posible escribir metacaracteres '&'
también es posible escribir metacaracteres &
gogh:~>
```

2.2.4 Escribiendo Scripts

Podemos almacenar en un fichero un conjunto de comandos del shell y a continuación darle permisos para ejecución. Si hacemos esto, hemos escrito un script. A estos ficheros se les llama también programas de shell. Para ejecutar un script sólo deberemos teclear el camino hasta su nombre[2] y a continuación pulsar la tecla return . Los script se utilizan para automatizar tareas que suelen ejecutarse de una forma más o menos regular y que por tanto pueden programarse. Cuando un script corre, el núcleo (kernel) del sistema operativo UNIX determina para que tipo de shell fue escrito (C

[2]En general esto no es necesario si el el directorio en el que se almacena el script se realizar una búsqueda cada vez que se ejecuta un comando. Aunque pareciera esta una tarea pesada, UNIX mantiene una lista de comandos a los que accede mediante hashing de una forma muy rápida como podremos apreciar

Shell, Bourne Shell, etc). Esto lo hace examinando la primera línea del script. Las reglas usadas son:

- Si el primer carácter de la primera línea es un `#`, es script será interpretado por en C Shell.
- Si la primera línea es de la forma `#!` *programa_ejecutable*, entonces el programa ejecutable *programa_ejecutable* es usado para interpretar el script.
- Si ninguna de las dos anteriores reglas es seguida, el script será interpretado por el Bourne Shell.

Si el carácter `#` aparece en cualquier otra parte que no sea como primer carácter del script, será interpretado como inicio de comentario, elementos que siempre deben de existir si es que deseamos que el mantenimiento del sistema no se convierta en una pesadilla. Es recomendable, que si vamos a usar el C Shell como shell para ejecutar un script, comencemos la primera línea con la cadena `#!` y de esa forma indiquemos claramente cual es el shell usado para ejecutar el script. A continuación se describe dos ejemplos de construcción de scripts. El primero interpretado por el C Shell y el segundo por el Korn Shell:

```
gogh:~> cat > script.csh (creamos el script)
# Este es un ejemplo de un script escrito para el C Shell.
echo -n La fecha de hoy es :
date    # imprimimos en pantalla (la salida estándar) la fecha de hoy.
^D
gogh:~> cat > script.ksh (creamos el script)
#! /bin/ksh
# Este es un ejemplo de un script escrito para el Korn Shell.
echo -n La fecha de hoy es :
date    # imprimimos en pantalla (la salida estándar) la fecha de hoy.
^D
gogh:~>chmod +x script.csh script.ksh (cambiamos los permisos para ejecución)
gogh:~> ls -lF script.csh script.ksh
-rwxrwxr--  1 vi      wheel        154 Apr 24 17:07 script.csh*
-rwxrwxr--  1 vi      wheel        180 Apr 24 17:09 script.ksh*
gogh:~> script.csh (ejecutamos el primer script)
La fecha de hoy es :Mon Apr 24 17:10:50 GMT 1995
gogh:~> script.ksh
La fecha de hoy es :Mon Apr 24 17:11:01 GMT 1995
gogh:~>
```

Como se puede observar en el ejemplo, lo primero que realizamos es ejecutar un **Child Shell** (shell hijo). Esto provoca que las instrucciones escritas en el script sean ejecutadas en un shell distinto al que lanzó el shell. Si este no se ejecuta en background, el **Parent Shell** (shell padre) duerme hasta que el script es ejecutado. Si el script se ejecuta en background, los dos shells se ejecutan simultáneamente.

2.2.5 Sub-Shells

Cuando entramos en el sistema haciendo el `login`, se nos asigna un shell inicial. Todos los comandos *simples* que ejecutemos serán interpretados por este shell. Pero existen ocasiones en las que el actual shell crea un sub-shell para realizar algunas tareas. Esto se produce:

- Cuando agrupamos un conjunto de comandos con los símbolos de la forma `(...)`, un shell hijo ejecuta los comandos entre paréntesis, mientras el shell padre duerme hasta que esto se produce. A continuación se muestra un ejemplo de esto.

```
gogh:~> pwd (vemos el directorio de trabajo)
/usr/users/v
gogh:~> (cd /; pwd) (agrupamos dos comandos)
/
gogh:~> pwd (al finalizar el sub-shell comprobamos que cd / no causa efecto)
/usr/users/v
gogh:~>
```

- Cuando un script es ejecutado, el shell padre crea un shell hijo que ejecuta los comandos del script. Si el script no es ejecutado en background, el shell padre duerme hasta que el script es ejecutado.

- Cuando un trabajo es ejecutado en backgound, el shell padre lanza un shell hijo que realiza esta tarea. En este caso, el shell padre sigue ejecutándose junto con el shell hijo.

2.2.6 Variables del Shell

Los Shells soportan dos clases de variables: **locales** al shell que estamos usando y que no están declaradas cuando abandonamos el shell y **globales** que permanecen activadas aunque nos cambiemos de shell. Por permanecer inalterables en cualquier entorno de shell normalmente las variables globales se llaman **variables de entorno**.

Los dos tipos de variables se almacenan en formato de cadenas. La única diferencia entre ellas es que cuando ejecutamos un sub-shell, sólo las de entorno son heredadas, y las locales no están definidas. El usuario puede definirse sus propias variables locales y de entorno.

A continuación mostramos una lista de variables de entorno comunes a todos los shell que encontramos.

Variable	**Contenido**
`$HOME`	El camino completo hacia el directorio de trabajo.
`$PATH`	Una lista con los directorios para la búsqueda de comandos.
`$MAIL`	El camino completo sobre el fichero de mail (**mailbox**)
`$USER`	La cadena que nos identifica como usuario del sistema.
`$SHELL`	El camino completo al ejecutable lanzado en el `login` shell.
`$TERM`	Cuál es el tipo de nuestro terminal.

La sintaxis de asignación de las variables depende del shell que estamos ejecutando, pero la forma en que son leídas es siempre la misma: debemos de escribir `$` seguido del nombre de la variable (normalmente en mayúsculas). Aquí tenemos algunos ejemplos de lectura de algunas variables:

```
gogh:~# csh (ejecutamos el C Shell)
# echo $USER (preguntamos por quien somos)
root (somos el super-usuario)
# echo $HOME (cual es nuestro directorio de entrada al sistema)
/root
# echo $TERM (cual es el tipo de nuestro terminal)
console
# echo $PATH (cuales son los caminos de búsqueda de comandos)
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/bin/X11:
/usr/TeX/bin:/usr/openwin/bin:/usr/games:.
# echo $MAIL (cual es nuestro directorio de correo)
/var/spool/mail/root
# echo USER=$USER, SHELL=$SHELL, TERM=$TERM (una consulta mas compleja)
USER=root, SHELL=/bin/bash, TERM=console
#
```

2.2.7 Terminando el shell

En todos los shells existe un comando que termina tanto los scripts como el shell (ya que una sesión interactiva con un shell puede considerarse como un script muy complejo, que se crea a la vez que se ejecuta el anterio comando). Este es:

exit *número*

Este comando termina todo aquello que lo ejecuta y retorna un código de salida que puede ser obtenido normalmente por un shell padre. En realidad casi todos los programas que podemos correr en UNIX devuelven códigos de salida que nos indican si estos se han ejecutado con éxito (devolviendo normalmente un 0). Esto posibilita las ejecuciones condicionales antes vistas. En el C shell, la variable `$status` contiene el valor retornado por el anterior comando ejecutado. Ejemplo:

```
iron% date (ejecutamos un comando con sin error)
Wed Apr 26 16:02:26 WET DST 1995
iron% echo $status (comprobamos que no ha habido error)
0
iron% cc prog.c (compilamos un programa que no existe)
cpp: prog.c: No such file or directory
iron% echo $status (este es el código de salida cuando no es posible compilar)
2
iron% awk (llamamos a la utilidad de búsqueda de cadenas sin parámetros)
awk: Usage: awk [-f source] [-Fc] ['cmds'] [variable=value ...] [file ...]
iron% echo $status (pero al no haber trabajado, devuelve su código de error)
2
iron%
```

De la misma forma, el comando exit puede retornar un código cuando es ejecutado. Ejemplo:

```
gogh:~> cat script_exit.csh (presentamos un pequeño script)
echo Este script retorna un código de salida de 3
exit 3 (que termina retornando el código de salida 3)
gogh:~> chmod +x script_exit.csh (asegurémonos de que es ejecutable)
gogh:~> script_exit.csh (ejecutamos el script)
Este script retorna un código de salida de 3 (salida del script)
gogh:~> echo $status (vemos el estado de salida)
3
```

2.3 El Bourne Shell

El Bourne Shell, escrito por Sthephen Bourne, fue el primer y popular shell para el sistema operativo UNIX y está disponible en todas las implementaciones realizadas de este sistema operativo[3]. Es programable y funcionalmente puede ser visto como un subconjunto del más potente Korn Shell (ksh), por lo que no es de extrañar que se le parezca mucho. El Bourne Shell sigue siendo muy importante en la construcción de scripts de arranque del sistema opertivo (realmente es el casi el único que se usa porque al ser el más básico asegura que los scripts de arranque siempre cumplan su cometido: arrancar el sistema).

2.3.1 Cuando el Bourne Shell arranca

El Bourne Shell es un programa escrito en lenguaje C y su ejecutable se almacena normalmente en `/bin/sh`. Este puede ser invocado de tres formas diferentes:

1. Cuando accedemos a la cuenta (login).
2. Cuando ejecutamos un script que lo invoca.
3. Tecleando el comando sh desde el shell que nos está atendiendo en ese momento (si es que la variable `$PATH` tiene el camino `/bin` definido, si no, deberemos escribir su camino completo).

EL Bourne Shell (y en realidad todos los shell que vamos a estudiar) puede funcionar de dos formas distintas: (1) de forma interactiva y (2) de forma no interactiva que se produce cuando lo invocamos desde un script. Estos son muy usados junto con el comando at que lanza el proceso especificado a una hora predefinida.

Cuando un Bourne Shell interactivo es ejecutado, este ejecuta el script `.profile` almacenado en el home directory (apuntado por la variable `$HOME`, si es que existe). Cuando termina de hacer esto muestra el prompt `$`, que puede ser cambiado modificando el valor de la variable `$PS1`. Un Bourne Shell no interactivo no ejecuta el fichero `$HOME\.profile`. Ejemplo de fichero de arranque `.profile`:

```
TERM=vt100                                        # Tipo de terminal usado
export TERM                                       # Copiamos la variable TERM al entorno
stty erase "^?" kill "^U" intr "^C" eof "^D"     # Metacaracteres usados
PATH='.:/bin:/usr/bin:/usr/ucb:/usr/local/bin'   # caminos de fichero ejecutables
```

2.3.2 Creando/asignando variables

La sintaxis de creación y asignación de contenido a una variable es:

{*nombre_variable*=*valor_variable*}+

Si la variable no existía será creada. Si existía, sólo se modifica su valor y el previo se pierde. Cualquier variable creada es local. Será de entorno si ya tenía esta cualidad antes de la asignación. Vamos algunos ejemplos de utilización:

[3] Curiosamente en Linux el shell sh no existe como tal, pués ha sido totalmente reemplazado por el bash . Esto lo notaremos si invomcamos la ayuda sobre el sh . Apreciaremos que el sistema operativo nos muestra la ayuda del bash .

```
$ nombre=Pepe apellido=Pepon edad=29
$ echo $nombre $apellido tiene $edad anhos
Pepe Pepon tiene 29 anhos
$ nombre=Pepe Pepon
Pepon: not found
$ nombre="Pepe Pepon"
$ echo $nombre
Pepe Pepon
$
$ verb=sing
$ echo I like $verbing
I like
$ echo I like ${verb}ing
I like singing
$
```

2.3.3 Leyendo una variable de la entrada estándar

Cuando necesitamos leer el valor de una variable desde el teclado cuando estamos ejecutando un script, el Bourne Shell dispone del comando:

read {*variable*}+

Veamos un ejemplo:

```
$ cat nombre.script
echo -n "Entra tu nombre: "
read nombre
echo Tu nombre es: $nombre
$ nombre.script
Entra tu nombre: Pepe Pepon
Tu nombre es: Pepe Pepon
$
```

2.3.4 Exportando variables

Es la forma en que varibles locales a shell se convierten en variales de entorno definidas fuera de este. La forma de hacerlo tiene la siguiente sintaxis:

export {*variable*}+

y la forma de ver el valor de las variables de entorno es la siguiente:

env {*variable=valor*}* *[comando]*

Como ejemplo:

```
$ export (preguntamos por las variables exportadas)
$ (pero no hay ninguna)
$ TERM=vt100 (definimos una variable local que define el tipo de terminal)
$ export (y volvemos a preguntar por las variables exportadas)
$ (pero sigue sin existir)
$ export TERM (exportamos la variable local)
$ export (y ahora si ha sido exportada)
export TERM
$ env (TERM aparece ahora entre las variables de entorno)
HOME=/usr/users/v
LOGNAME=vruiz
SHELL=/bin/csh
TERM=vt100
USER=vruiz
```

2.3.5 Variables de solo lectura

El Bourne Shell dispone de un comando para proteger el valor de variables de forma que sólo puedan ser leidas:

readonly {*variable*}*

En el siguiente ejemplo se proteje una variable local, que cuando pasa se convierte en variable de entorno y existe en un nuevo sub-shell, deja de estar protejida:

```
$ nombre=Pepe (creamos una variable local)
$ echo $nombre (presentamos el valor de la variable creada)
Pepe
$ readonly nombre (protegemos la variable)
$ readonly (preguntamos por las variables protegidas)
readonly nombre nombre (es la única variable protegida)
$ nombre=Pepone (intentamos modificarla)
nombre: is read only (pero no podemos)
$ export nombre (la convertimos en variable de entorno)
$ echo $nombre (y su valor no se ha modificado)
Pepe
$ nombre=Pepone (por ser de entorno sigue estando protegida)
nombre: is read only
$ sh (lanzamos un sub-shell)
$ readonly (preguntamos por las variable protejidas)
$ echo $nombre (y la variable sigue inalterada)
Pepe
$ nombre=Pepone (pero puede ser modificada porque es una copia)
$ echo $nombre
Pepone
$ exit (salimod del sub-shell)
$ echo $nombre (y la variable está como la dejamos)
Pepe
$
```

2.3.6 Aritmética

Aunque el Bourne Shell no soporta operaciones aritméticas directamente éstas si son soportadas usando el comando

expr *expresión*

que evalúa la expresión formulada y envía el resultado a la salida estándar. Este valor puede ser así asignado a una variable. Veamos primero los operadores utilizados ordenados por prioridad:

<table>
<tr><th>Operador</th><th>Significado</th></tr>
<tr><td><code>* / %</code></td><td>multiplicación, división y resto</td></tr>
<tr><td><code>+ -</code></td><td>suma y resta</td></tr>
<tr><td><code>= \> \>= \< \<= !=</code></td><td>operadores de comparación</td></tr>
<tr><td><code>\&</code></td><td>AND lógica</td></tr>
<tr><td><code>\|</code></td><td>OR lógica</td></tr>
<tr><td>match cadena expresión</td><td>Busca la cadena en la expresión</td></tr>
<tr><td>substr cadena pos1 pos2</td><td>Extrae los caracteres cadena[pos1]...cadena[pos2]</td></tr>
<tr><td>index cadena subcadena</td><td>Busca la posición de la subcadena en la cadena cadena</td></tr>
<tr><td>length cadena</td><td>Retorna la longitud de la cadena</td></tr>
</table>

Para modificar las prioridades podemos usar los paréntesis `\(` y `\)`. Veamos algunos ejemplos:

```
$ x=1
$ x=`expr $x + 1` (una sencilla suma)
$ echo $x
2
$ x=`expr 2 + 3 \* 5` (ahora junto con una multiplicación)
$ echo $x 17
$ echo `expr \( 2 + 3 \) \* 5` (el uso de los paréntesis)
25
$ echo `expr length "cat"` (calculando la longitud de una cadena)
3
$ echo `expr substr "hola caracola" 5 5 (extrayendo una subcadena ...)
> ` (... pero se me olvida cerrar la comilla que agrupa la expresión)
cara
$ echo `expr \(4\>5\)` (los espacios en blanco son muy importantes)
(4>5) (pues el resultado obtenido no es correcto)
$ echo `expr \( 4 \> 5 \)`
0 (4 no es mayor que 5)
$
```

Todos estos ejemplos debieran funcionar sin modificación alguna usando bash .

2.3.7 Expresiones condicionales

La forma de testear si una determinada expresión es verdadera o falsa se realiza en el Bourne Shell con el comando

test *expresión*

que retorna cero si la expresión es verdadera y otro valor diferente de cero si la expresión es falsa. Este valor devuelto se utiliza en la mayoría de las ocasiones para realizar un salto. A continuación se muestra de forma detallada todas las acciones de este comando.

Forma	Significado
-b *fichero*	Verdadero si *fichero* es un fichero especial de bloques.
-c *fichero*	Verdadero si *fichero* es un fichero especial de caracteres.
-d *fichero*	Verdadero si *fichero* es un directorio.
-f *fichero*	Verdadero si *fichero* no es un directorio.
-g *fichero*	Verdadero si *fichero* es un fichero sed-group-id.
-h *fichero*	Verdadero si *fichero* es un enlace simbólico.
-k *fichero*	Verdadero si *fichero* es un sticky bit set.
-p *fichero*	Verdadero si *fichero* es un pipe.
-r *fichero*	Verdadero si *fichero* puede ser leído.
-s *fichero*	Verdadero si *fichero* contiene al menos un carácter.
-u *fichero*	Idem que -g.
-w *fichero*	Verdadero si *fichero* es modificable.
-x *fichero*	Verdadero si *fichero* es ejecutable.
-t *descriptor*	Verdadero si el *descriptor* está asociado a un terminal.
-l *cadena*	Retorna la longitud de la *cadena*.
-n *cadena*	Verdadero si *cadena* tiene al menos un carácter.
-z *cadena*	Verdadero si la *cadena* está vacía.
`cad1=cad2`	Verdadero si `cad1` es igual a `cad2`.
`cad1!=cad2`	Verdadero si `cad1` es distinta de `cad2`.
`cadena`	Verdadero si la cadena no es `null`.
`num1 -eq num2`	Verdadero si el entero `num1` es igual a `num2`.
`num1 -ne num2`	Verdadero si el entero `num1` no es igual a `num2`.
`num1 -gt num2`	Verdadero si el entero `num1` es mayor que `num2`.
`num1 -ge num2`	Verdadero si el entero `num1` es mayor o igual que `num2`.
`num1 -lt num2`	Verdadero si el entero `num1` es menor que `num2`.
`num1 -le num2`	Verdadero si el entero `num1` es menor o igual que `num2`.
`!expresión`	Verdadero si la `expresión` es falsa.
`expr1 -a expr2`	Verdadero si ambas expresiones son verdaderas.
`expr1 -o expr2`	Verdadero si una de las expresiones es verdadera.
`\( expr \)`	Forma de agrupar expresiones.

Ejemplos de uso de estas comparaciones se usaran en el resto de ejemplos incluidos en el uso de las estructuras de control que basan su funcionamiento en una comparación de esta clase.

2.3.8 Estructuras de control

Como ya indicamos el Bourne Shell es muy adecuado para construir scripts de arranque y mantenimiento del sistema operativo. Esto significa que debe de ser muy programable y por lo tanto es rico en estructuras de control que son estudiadas a continuación:

case ... in ... esac

Así el shell soporta el salto multicamino en función de una única comprobación de una cadena. Su sintaxis:

> **case** *expresión* **in**
> *patrón* {— *patrón*}*
> *lista*

;;
esac

Veamos un ejemplo:

```
$ cat case.sh
#! /bin/sh
echo menu test program
echo
stop=0                                      # flag de salida del bucle
while test $stop -eq 0                      # el bucle
# cat coge como entrada estándar lo escrito entre las etiquetas
# END_OF_MENU, líneas que no pueden ir comentadas.
do
    cat << END_OF_MENU
    1:   print the date
    2,3: print the current working directory
    4;   exit
END_OF_MENU
    echo
    echo -n 'your choice ... '
    read reply                                # leemos la entrada estándar
    echo
    case $reply in
      "1")
        date
        ;;
      "2"3")
        pwd
        ;;
      "4")
        stop=1
        ;;
      *)
      echo illegal choice
      ;;
    esac
done
$ case.sh
menu test program

    1:   print the date
    2,3: print the current working directory
    4;   exit

your choice ... 1

Wed Jul 17 18:59:41 GMT+0100 1996
    1:   print the date
    2,3: print the current working directory
    4;   exit

your choice ... 5

illegal choice      1:   print the date
    2,3: print the current working directory
    4;   exit

your choice ... 4

$
```

for ... do ... done

El comando **for** permite que una serie de comandos puedan ser ejecutados un número determinado de veces variando la variable que controla el lazo. Esta es su sintaxis:

> **for** *variable [* **in** *{palabra}*]*
> **do**
> *lista de comandos*
> **done**

Veamos un ejemplo:

```
$ cat for.sh
for color in red yellow green blue
do
echo one color is $color
done
$ for.sh
one color is red
one color is yellow
one color is green
one color is blue
$
```

if ... then ... fi

Esta estructura de control permite ejecutar porciones de código de scripts condicionalmente. Su sintaxis:

> **if** *expresión1* **then**
> *lista de comandos1*
> **elif** *expresión2* **then**
> *lista de comandos2*
> **else** *lista de comandos3*
> **fi**

Un ejemplillo:

```
$ cat if.sh
echo -n énter a number: '
read number
if [ $number -lt 0 ]
then
    echo negative
elif [ $number -eq 0 ]
then
    echo zero
else
    echo positive
fi
$ if.sh
enter a number: 0
zero
$ if.sh
enter a number: -4
negative
$ if.sh
enter a number: 10
positive
$
```

trap

El comando **trap** permite ejecutar un comando especificado cuando se produce un determinado evento (recibe una señal con un determinado valor). Aquí está su sintaxis:

trap *[[comando] {señal}+]*

Ejemplo:

```
$ cat trap.sh
trap 'echo Control-C; exit 1' 2
while (test 1 -eq 1) # También podemos escribir [ 1 -eq 1 ] como condidión
do
     echo infinite loop
     sleep 2
done
$ trap.sh
infinite loop
infinite loop
^C
Control-C
$
```

Para ver todas las señales a las que atiende `trap`, dar la opción `-l`. Para más información acerca de las señales, hay que consultar el fichero `/usr/include/signal.h` o invocar a `man signal`.

until ... do ... done

Sirve para realizar lazos. Su sintaxis:

until *expresión*
do
serie de comandos
done

Este comando ejecuta una *serie de comandos* mientras se cumpla la *expresión* sea verdadera. Ejemplo:

```
$ cat until.sh
x=1
until [ $x -gt 3 ]
do
     echo x=$x
     x=`expr $x + 1`
done
$ until.sh
x=1
x=2
x=3
$
```

while ... do ... done

Es similar al anterior comando y se utiliza para construir lazos. Su sintaxis:

while *expresión*
do
serie de comandos
done

Así se ejecuta la *serie de comandos* si la *expresión* es verdadera. Ejemplo:

```
$ cat while.sh
x=1
while [ $x -le $1 ]
do
    y=1
    while [ $y -le $1 ]
    do
      echo -n `expr $x \* $y` "    "
      y=`expr $y + 1`
    done
    echo
    x=`expr $x + 1`
done
$ while.sh
while.sh: test: argument expected
$ while.sh 7
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
$
```

2.4 El C Shell

El **C Shell** fue desarrollado posteriormente al Bourne shell y está pensado en facilitar el control del sistema al programador en lenguaje C. La razón de esto es que su sintaxis, como vamos a apreciar es muy similar a la de este lenguaje. En este sección estudiaremos aquellas sentencias de control más usadas y aprenderemos a escribir **scripts** para $\boxed{\text{csh}}$.

2.4.1 Cuándo el C shell arranca

El C Shell es un programa ejecutable escrito en lenguaje C (como casi todo lo que tiene que ir rápido en UNIX) almacenado con el nombre de `/bin/csh`. Existen 3 formas distintas de ejecutarlo:

- Durante el `login`.
- En la ejecución de un script.
- Escribiendo $\boxed{\text{/bin/csh}}$ en la línea de comandos. Si la variable `$PATH` tiene el camino `/bin` definido, con sólo teclear $\boxed{\text{csh}}$ es suficiente.

La secuencia de arranque lee y ejecuta los comandos almacenados en 3 ficheros que se detallan a continuación:

Paso	**Forma de invocar**	**Acción**
1	en cualquier caso	Ejecutar script `$HOME/.cshrc` si exite.
2	en el `login`	Ejecutar script `/etc/login` si existe.
3	en el `login`	Ejecutar script `$HOME/.login` si existe .

Cuándo el C Shell está corriendo suele mostrar el carácter `%` o el carácter `>`, aunque cómo veremos a continuación esto es poco definitivo. Un fichero `.login` contiene definiciones de variables que son necesarias para la ejecución del shell. Por ejemplo:

```
echo -n "Entra el tipo de terminal (por defecto vt100): "
set termtype = $< (leemos lo siguiente tecleado por el usuario)
set term = vt100 (valor por defecto)
if($termtype != ) set term = $termtype (valor tecleado)
unset termtype (destruye la variable termtype)
set path=(. /usr/bin /usr/local/bin /bin) (camino de búsqueda de ejecutables)
stty erase "^? "kill "^U"intr "^C"eof "^D"crt crterase (controles)
set cdpath = (~) (camino de b'squeda del comando chdir, que funciona igual que cd)
set history = 40 (numero de comandos recordados)
set prompt = "! % "(numero de comando, space, %, space)
set savehist = 32 (numero de comandos recordados entre sesiones)
```

generaría el siguiente `login` en Linux:

```
Welcome to Linux 1.0.8.
gogh login: guess
Last login: Tue Apr 25 02:00:57 on tty4
Linux 1.0.8. (Posix).
Entra el tipo de terminal (por defecto vt100): (pulsamos return )
1 %
```

En la variable `$prompt` podemos declarar los siguientes metacaracteres que significan:

Metacaracteres	Significado
`!`	Número de comando que se esta construyendo.
`%m`	Nombre de la máquina que estamos utilizando.
`$USER`	Nombre del usuario en la maquina.
`%`	directorio actual en el que estamos situados.
`%@`	Hora del sistema.
`%#`	Indica si entramos como super-usuario:
	`>`: somos un usuario normal.
	`#`: somos el super-usuario.

El fichero `.cshrc` contiene generalmente comandos tales como los de **alias** y aquellas variables (locales o de entorno) que necesitemos para nuestro trabajo. El complejo nombre que obtiene se debe a las siglas de SHell Run Comands". Como ejemplo de `.cshrc` valga el siguiente:

```
alias emacs /usr/local/bin/emacs (para ejecutar emacs rápidamente)
alias h history (teclear history es tedioso)
alias ll ls -l
alias ls ls -F
alias rm rm -i
alias m more
```

que provoca que podamos utilizar comandos de la forma:

```
20 % h  (lo mismo que teclear history)
       1  1:56    ls
       2  1:56    pwd
       3  1:56    ls -la
       4  1:57    more .login
       5  2:01    history
       6  2:01    vi .csrhc
       7  2:03    csh
       8  2:03    ls -l
       9  2:03    ls -la
      10  2:03    mv .csrhc cshrc
      11  2:03    csh
      12  2:03    ls -la
      13  2:03    mv cshrc .cshrc
      14  2:04    csh
      15  2:04    cat .cshrc
      16  2:17    ls -l
      17  2:17    cat .login
      18  2:19    exit
      19  2:22    cat .cshrc
      20  2:23    h
%
```

Manipulación de las variables

La creación de las variables se realiza de la siguiente forma:

```
# set Nombre=Vicente (creamos una variable local llamada Nombre)
# set Apellidos='Gonzalez Ruiz'(otra variable local)
# echo $Nombre $Apellidos (comprobamos como están las variables)
Vicente Gonzalez Ruiz
# csh (invocamos un sub-shell tipo C)
# echo $Nombre $Apellidos (las variables ?)
Nombre: Undefined variable (no están definidas !, lógico, son locales)
# exit (abandonamos el sub-shell)
exit
# echo $Nombre $Apellidos (volvemos a preguntar por las variables)
Vicente Gonzalez Ruiz (son locales a este shell)
# setenv Nombre Vicente (ahora declaramos la variable de entorno)
# setenv Apellidos 'Gonzalez Ruiz'(las locales son destruidas)
# echo $Nombre $Apellidos (pues tienen el mismo nombre)
Vicente Gonzalez Ruiz (contenido de las variables de enntorno)
# csh (invocamos el C Shell)
# echo $Nombre $Apellidos (preguntamos por las variables de entorno)
Vicente Gonzalez Ruiz (y siguen estando activas en el sub-shell)
# exit
exit
# echo $Nombre $Apellidos (y también en el shell de partida)
Vicente Gonzalez Ruiz
#
```

Como hemos podido apreciar, el C Shell soporta variables locales y de entorno. Las locales pueden tener 1 o o varias asignaciones simultánemamente (una lista de valores) mientras que las de entorno no.

Para asignar un valor simple a una variable local se utiliza el comando:

set {*variable[=valor]*}*

Si no argumentamos (escribimos set y pulsamos enter) se muestra el contenido de todas las variables locales definidas hasta ese momento. Si no damos *valor*, borramos (no destruimos) la variable. Ejemplos:

```
33 % set flag (borramos la variable local flag)
34 % echo $flag (comprobamos que esta borrada)
35 % set color = red (creamos una variable color que contiene red)
36 % echo $color (vemos el contenido de color)
red
37 % set name = Vicente Gonzalez (creamos una variable no muy bien)
38 % echo $name
Vicente
39 % set name = 'Vicente Gonzalez'(ahora si la creamos bien)
40 % echo $name
Vicente Gonzalez
41 % set (listamos todas las variables locales)
argv ()
cdpath              /home/guess
color               red
name                Vicente Gonzalez
term                vt100
user                guess
42 %
```

2.4.2 Accediendo a las variables

Además del simple acceso explicado hasta ahora (`$variable`), el C Shell soporta los siguientes accesos más complejos:

Sintaxis	Acción
`${variable}`	Extrae el contenido de la variable. (ver ejemplo)
`${?variable}`	Vale 1 si la variable está creada, 0 en otro caso.

En los siguientes ejemplos ponemos de manifiesto la utilildad de estos métodos de acceso. En el primer caso, nos permite concatenar el contenido de la variable y una cadena, y en el segundo, nos permite usar variables con contenidos no numéricos como condición de comparación:

```
35 % set grupo = Motor
36 % echo Me gusta la musica de los $grupoHead
grupoHead: Undefined variable.
37 % echo Me gusta la musica de los ${grupo}Head
Me gusta la musica de los MotorHead
38 %
39 % cat > flag.csh
# Creamos una variable llamada flag pero sin un valor en concreto
set flag
# Preguntamos por si la varible flag ha sido creada
if ($?flag) then
echo La variable local flag ha sido creada
endif
40 % chmod +x flag.csh
41 % flag.csh
La variable local flag ha sido creada
42 %
```

2.4.3 Trabajando con conjuntos de valores

En C Shell es posible asignar una lista de valores a una variable local. La sintaxis es la siguiente:

set *variable=({valor}*)}**

Por ejemplo:

```
44 % set colores = (rojo amarillo verde) (una variable con 3 valores)
45 % echo $colores
rojo amarillo verde
46 %
```

2.4.4 Accediendo a conjuntos de valores

El C Shell trabaja con este tipo de variables como en el caso de los más puros arrays en el lenguaje C:

Sintaxis	Acción
`$ variable[selector]` `${variable[selector]}`	Accede a un subconjunto de la lista de valores que pueda tener la variable (ver ejemplo).
`$#variable` `${#variable}`	Averigúan el cardinal de la variable.

Ejemplo:

```
44 % set colores = (rojo amarillo verde)
45 % echo $colores
rojo amarillo verde
46 % echo $colores[1] (accedemos al primer valor)
rojo
47 % echo $colores[2-3] (accedemos al segundo y al tercero)
amarillo verde
48 % echo $colores[4] (accedemos a un valor no definido)
colores: Subscript out of range.
49 % echo $#colores (cuantos valores tiene la variable)
3
50 %
```

2.4.5 Incrementando el tamaño de un conjunto

Para añadir un elemento a la lista o modificar uno de ellos se realiza lo siguiente:

```
 50 % !44 (ejecutamos el comando 44)
set colores = ( rojo amarillo verde ) (este es el comando 44)
51 % set colores[4] = rosa (el cuarto valor va a ser el color rosa)
set: Subscript out of range. (pero no hay sitio en la variable)
52 % set colores = ( $colores azul ) (asi ampliamos el rango)
53 % echo $colores
rojo amarillo verde azul
54 % set colores[4] = rosa (ahora si el cuarto valor va a se el color rosa)
55 % !53 (ejecutamos el comando 53)
echo $colores
rojo amarillo verde rosa
56 % set otros_colores = ( negro magenta ) (otro conjunto de colores)
57 % set colores = ( $colores $otros_colores ) (concatenamos los conjuntos)
58 % echo $colores
rojo amarillo verde rosa negro magenta
59 %
```

2.4.6 Variables locales predefinidas

El C Shell normalmente (no siempre están definidas, dependiendo de la versión del shell y del sistema operativo) tiene las siguiente variables creadas y operativas:

Variable	Valor
`$?0`	1 si el Shell ejecuta comandos de un fichero, 0 en otro caso.
`$<`	La siguiente línea (terminada con `newline`)de la entrada estándar.
`$argv`	Una lista conteniendo todos los parámetros de llamada: `argv[1]=$1`.
`$cdpath`	Una lista alternativa de directorios que el comando *chdir* usa para propósitos de búsqueda.
`$cwd`	El directorio actual de trabajo.
`$echo`	Indica si hay eco. Se activa si la opción -x del comando csh fue activada.
`$history`	El tamaño de la lista de historia de comandos en una sesión.
`$home`	El directorio de entrada tras el `login`.
`$ignoreeof`	Previene al C Shell de la terminación con ˆD .
`$mail`	Lista con los archivos chequeados por la utilidad de `mail`.
`$notify`	Si no está definida, se notifica cualquier cambio en el estado de un proceso.
`$path`	Usada para localizar los ficheros ejecutables.
`$prompt`	Almacena el prompt del shell.
`$savehist`	Número máximo de comandos recordados en la lista de comandos entre sesiones.
`$shell`	Camino completo hacia el fichero ejecutable del shell.
`$status`	El código de salida del anterior comando ejecutado.
`$time`	Tiempo máximo que transcurre sin que un proceso muestre sus estadísticas.
`$verbose`	Activa si la opción -v del comando csh fue activada.

2.4.7 Destruyendo variables locales

Para liberar la memoria asignada a variables locales que ya no van a ser utilizadas o para eliminar la influencia de variables locales que basan su funcionamiento no en su contenido, sino únicamente en si están o no declaradas, podemos usar el comando:

1. **unset** *variable*

Por ejemplo:

```
17 gogh vi  4:01pm > set a = 2
18 gogh vi  4:01pm > echo $a
2
19 gogh vi  4:01pm > unset a
20 gogh vi  4:01pm > echo $a
a: Undefined variable.
21 gogh vi  4:01pm >
```

2.4.8 Las variables de entorno

Para asignar un valor a una variable de entorno, se usa el siguiente comando:

1. **setenv** *variable valor*

Si el nombre de la variable no existe anteriormente, esta es automáticamente creada; en otro caso, es sobreescrita. Las variables de entorno no pueden contener lista de valores. Un ejemplo:

```
gogh login: guess (entramos a la cuenta como invitado)
Last login: Tue Apr 25 15:56:39 on tty2
Linux 1.0.8. (Posix).
Entra el tipo de terminal (por defecto vt100): (pulsamos [return] )
33 % echo $TERM (visualizamos el tipo de terminal)
vt100
34 % setenv TERM vt52 (intentamos cambiarlo a vt52)
tcsh: No entry for terminal type "vt52"(pero ese no es soportado)
tcsh: using dumb terminal settings.
35 % setenv TERM console (probamos con el tipo console)
36 % echo $TERM (esta vez si tenemos exito)
console
37 %
```

La única varible de entorno predefinida en el C Shell es `$LOGNAME` que almacena el nombre del usuario en el sistema:

```
37 % echo $LOGNAME
guess
38 %
```

Como se ha podido observar, por convenio las variables locales normalmente se escriben en minúscula y las de entorno en mayúscula aunque por supuesto podría haber sido al revés.

2.4.9 Destruyendo variables de entorno

Similarmemte a como destruimos las variables locales, las de entorno liberan su memoria asignada mediantes el comando

unsetenv *variable*

Por ejemplo:

```
20 gogh vi  4:01pm > echo $a
a: Undefined variable.
21 gogh vi  4:01pm > setenv a 2
22 gogh vi  4:05pm > echo $a
2
23 gogh vi  4:05pm > unsetenv a
24 gogh vi  4:05pm > echo $a
a: Undefined variable.
25 gogh vi  4:05pm >
```

2.5 Las expresiones en el C Shell

2.5.1 Operadores de cadena

El C Shell soporta los siguientes operadores de cadena:

Operador	Significado
==	Retorna verdadero si las dos variables almacenan la misma cadena.
!=	Retorna verdadero si las variables almacenan cadenas distintas .
=~	Como ==, pero el operando de la derecha puede contener *wildcards*.
!~	Como !=, pero el operando de la derecha puede contener *wildcards*.

Como ejemplo, adjuntamos un script que identifica varias palabras distintas como 'yes':

```
38 % cat > expr1.csh
#!/bin/csh (nos aseguramos de que es el csh quien ejecuta este script)
echo -n "do you like the C Shell ?" # Preguntamos: le gusta el C Shell
set respuesta = $<                  # Leemos la contestación del usuario
if ($respuesta == "yes") then
          echo Has entrado yes
else if ($respuesta =~ y*) then     # Buscamos palabras que comienzan por (y)
          echo Suponemos que dices yes
endif
39 % chmod +x expr1.csh
40 % expr1.csh
do you like the C Shell ?yeah
Suponemos que dices yes
41 %
```

2.5.2 Expresiones aritméticas

Se soportan los siguientes operadores aritméticos por orden de precedencia:

Operador	**Significado**
`++ --`	Autoincremento y autodecremento.
`-`	Menos (cambio de signo).
`!`	Negación lógica.
`* / %`	Multiplicación, división y resto.
`+ -`	Adición, substracción.
`<< >>`	Desplazamiento bit a bit hacia la izquierda y hacia la derecha.
`<= >= < >`	Menor o igual, mayor o igual, menor y mayor.
`== !=`	Igualdad, distinto.
& $\wedge\vert$	AND, XOR y OR bit a bit.
$\vert\vert$ &&	OR lógica y AND lógica.

y como ejemplo:

```
43 % cat > expr3.csh
#/bin/csh
set a = 3
set b = 5
if($a>2 && $b>4) then
          echo Pues si, esto parece que funciona
endif
44 % chmod +x expr3.csh
45 % expr3.csh
Pues si, esto parece que funciona
46 %
```

Para asignar el valor de una expresión a una variable usamos el operador `@`:

```
52 % set a = 2 * 2
set: Variable name must begin with a letter.
53 % @ a = 2 * 2
54 % echo $a
4
55 % @ a = $a + $a
56 % echo $a
8
57 % set flag = 1
58 % @ b = ($a && $flag)
59 % echo $b
1
60 % @ b ++
61 % echo $b
2
62 %
```

2.5.3 Expresiones para la gestión de ficheros

Para trabajar con fichero, el C Shell posee un potente repertorio de operadores que trabajan de la forma `-w` *nombre_del_fichero*. En la siguiente tabla son descritos:

Operador	Significado
`-r`	Averiguamos si tenemos permiso de lectura.
`-w`	Averiguamos si tenemos permiso de escritura.
`-x`	Averiguamos si tenemos permiso de ejecución.
`-e`	Averiguamos si el fichero existe.
`-o`	Averiguamos si somos dueño del fichero.
`-z`	Averiguamos si el fichero existe, pero tiene 0 bytes.
`-f`	Averiguamos si se trata de un fichero regular (no directorio o fichero especial).
`-d`	Averiguamos si se trata de un directorio.

Y a continuación se expone un script que determina si un fichero es modificable o no:

```
62 % cat > expr4.csh
#/bin/csh
echo -n "Dame el nombre del fichero que quieres borrar o modificar: "
set filename = $<              # Obtenemos la entrada estándar
if(! (-w "$filename")) then    # Chequeamos el derecho de escritura
          echo No tienes permiso
else
          echo Si tienes permiso. Adelante !
endif
63 % chmod +x expr4.csh
64 % ls -l
total 5
-rw-r--r--  1 guess    users         527 Apr 25 02:48 1
-rwxr-xr-x  1 guess    users         312 Apr 25 18:44 expr1.csh*
-rwxr-xr-x  1 guess    users          90 Apr 25 18:55 expr3.csh*
-rwxr-xr-x  1 guess    users         259 Apr 25 19:21 expr4.csh*
-rwxr-xr-x  1 guess    users         199 Apr 25 16:03 flag.csh*
65 % expr4.csh
Dame el nombre del fichero que quieres borrar o modificar: 1
Si tienes permiso. Adelante !
66 % expr4.csh
Dame el nombre del fichero que quieres borrar o modificar: /bin/csh No tienes permiso 67 %
```

2.6 Los aliases en el C shell

Los *aliases* has sido creados para tener que escribir menos, e incluso como ahora veremos, equivocándonos.

2.6.1 Creando un alias

Usado normalmente en el script `.cshrc`, el comando para crear alias es el siguiente:

alias *[palabra [cadena]]*

Y unos cuantos ejemplos para ver la utilidad:

```
33 % alias dir 'ls -aF'(construimos el dir del MS-DOS)
34 % dir
./          .emacs      .less       .term/      expr3.csh*
../         .history    .lessrc     1           expr4.csh*
.cshrc      .kermrc     .login      expr1.csh*  flag.csh*
35 % dir *.c
ls: No match.
36 % dir *.csh
expr1.csh*  expr3.csh*  expr4.csh*  flag.csh*
37 % alias dir (mostramos el contenido del alias)
ls -aF
38 % alias ls-l 'ls -l'(por si nos equivocamos al escribir 'ls -l')
39 % ls-l
total 5
-rw-r--r--  1 guess    users         527 Apr 25 02:48 1
-rwxr-xr-x  1 guess    users         312 Apr 25 18:44 expr1.csh*
-rwxr-xr-x  1 guess    users          90 Apr 25 18:55 expr3.csh*
-rwxr-xr-x  1 guess    users         259 Apr 25 19:21 expr4.csh*
-rwxr-xr-x  1 guess    users         199 Apr 25 16:03 flag.csh*
40 %
```

2.6.2 Destruyendo un alias

Para destruir un alias, usaremos el comando:

unalias *palabra*

Ejemplo:

```
40 % unalias ls-l
41 % ls-l
ls-l: Command not found.
42 %
```

2.7 La historia de comandos en el C Shell

El C Shell mantiene una lista con los últimos comandos tecleados. Estos pueden ser editados y re-ejecutados para facilitar la existencia al sufrido usuario. El responsable todo esto es el metacarácter `!`. La cantidad de entradas recordadas depende del valor almacenado en la variable local `$history` y el número de entradas salvadas de una sesión a otra se almacena en la variable local `$savehist`. Para obtener la historia de los comandos ejecutados hasta ese instante, teclearemos:

history *[-rh][número]*

La opción -r presenta la lista en orden inverso y la opción -h la presenta sin numerar. Ejemplo:

```
45 % history 4
      42  23:16   history -h
      43  23:17   ls -r
      44  23:17   history -r
      45  23:22   history 4
46 %
```

2.7.1 Re-ejecución de comandos

Para ejecutar uno de los comandos del fichero de historia tenemos las siguientes formas:

Forma	Acción
!!	Extrae el último comando.
!*número*	Extrae el comando indicado.
!*prefijo*	Extrae el último comando que posee el *prefijo* indicado.
!?*subcadena*?	Extrae el último comando que posee la *subcadena* indicada.

Ejemplos:

```
45 % history 4 (los 4 ultimos comandos)
      42  23:16   history -h
      43  23:17   ls -r
      44  23:17   history -r
      45  23:22   history 4
      46 % !! (ejecutamos el ultimo comando)
46 % history 4
      43  23:17   ls -r
      44  23:17   history -r
      45  23:22   history 4
      46  23:46   history 4
47 % !43 (ejecutamos el comando 43)
ls -r
flag.csh*   expr4.csh*  expr3.csh*  expr1.csh*  1
48 % !l (ejecutamos el ultimo comando que empieza por l)
ls -r
flag.csh*   expr4.csh*  expr3.csh*  expr1.csh*  1
49 % !?chmod? (ejecutamos el último comando que contiene la cadena chmod)
chmod +x expr4.csh
50 %
```

2.8 Las estructuras de control en el C Shell

El C Shell soporta tiene un potente repertorio de sentencias de control para la escritura de complejos y funcionales scripts. En esta sección se estudian las más importantes:

2.8.1 Agrupando ficheros

Existe una sentencia llamada foreach que permite ejecutar una lista de comandos para cada uno de los valores de una lista. La sintaxis es:

foreach *variable (lista_de_valores)*
lista de comandos a ejecutar
end

Ejemplo:

```
50 % foreach color (rojo amarillo verde)
foreach? echo Un color es $color
foreach? end
Un color es rojo
Un color es amarillo
Un color es verde
51 %
```

2.8.2 Salto incondicional

El comando $\boxed{\text{goto}}$ permite que saltemos incondicionalmente hasta la línea etiquetada. La sintaxis:

goto *etiqueta*

Ejemplo:

```
72 % cat goto.csh
#!/bin/csh (nos aseguramos de estar ejecutando el C Shell)
echo Hola
goto fin (saltamos a la línea siguiente a la línea etiquetada con fin)
echo esto no será escrito
fin:
echo caracola (saltamos a esta línea)
73 % goto.csh
Hola
caracola
74 %
```

2.8.3 Salto condicional

La sentencia de control del flujo condicional tiene dos versiones:

if *(expresión) comando*

Ejemplo:

```
74 % if (5 > 3) echo Cinco es mayor que tres
Cinco es mayor que tres
75 %
```

Y la versión:

if *(expresión1)* **then**
bloque de comandos
else if *(expresión2)* **then**
bloque de comandos
else
bloque de comandos
endif

Ejemplo:

```
83 % cat if.csh
#!/bin/csh
echo -n 'Teclea un numero: '
set numero = $<
if ($numero < 0) then
          echo El numero es negativo
else if ($numero == 0) then
          echo El numero es cero
else
          echo El numero es positivo
endif
84 % if.csh
Teclea un numero: 0
El numero es cero
85 % if.csh
Teclea un numero: 433
El numero es positivo
86 % if.csh
Teclea un numero: -1
El numero es negativo
87 %
```

2.8.4 Lazos

Es muy sencillo repetir varias veces un comando (o un script). La sintaxis:

repeat *expresión_arimética comando*

Ejemplo:

```
87 % repeat 3 echo 'Hola Caracola'(las comillas se pueden omitir)
Hola Caracola
Hola Caracola
Hola Caracola
88 %
```

Existe otra sentencia de control similar:

while *(expresión_aritmética)*
lista de comandos
end

Ejemplo:

```
102 % cat while.csh
#!/bin/csh (sobre el C Shell)
set x = 1 (una variable local)
while ($x <= $1) (mientras x ¡= primer parametro dado)
set y = 1
     while ($y <= $1)
       @ v = $x * $y (la tabla de multiplicar)
       echo -n $v "          " (un tabulador)
       @ y ++
     end
     echo ""
     @ x ++
end
103 % while.csh 7
1       2        3        4        5        6        7
2       4        6        8       10       12       14
3       6        9       12       15       18       21
4       8       12       16       20       24       28
5       10      15       20       25       30       35
6       12      18       24       30       36       42
7       14      21       28       35       42       49
104 %
```

2.8.5 Condicional múltiple

También disponemos de una sentencia de control multicamino, en función de una única expresión.

switch *(expresión)*
case *patrón1:*
lista de comandos
breaksw
case *patrón2:*
case *patrón3:*
lista de comandos
breaksw
default:
lista de comandos por defecto
endsw

Ejemplo:

```
113 % cat menu.csh
#!/bin/csh
echo  PROGRAMA MENU
set stop = 0                # flag de terminación del programa
while ($stop == 0)
          cat << ENDOFMENU
          1    : Imprimir la fecha.
          2, 3 : Imprimir el directorio de trabajo.
          4    : Salir
ENDOFMENU
echo ""
echo -n 'Tu elección ? '
set reply = $<
echo ""
switch ($reply)
    case "1":
      date
      breaksw
    case "2":
    case "3":
      pwd
      breaksw
    case "4":
      set stop = 1
      breaksw
    default:
      echo Elección ilegal
      breaksw
    endsw
end
114 % menu.csh
PROGRAMA MENU
          1    : Imprimir la fecha.
          2, 3 : Imprimir el directorio de trabajo.
          4    : salir
Tu eleccion ? 1
Wed Apr 26 01:03:18 GMT 1995
          1    : Imprimir la fecha.
          2, 3 : Imprimir el directorio de trabajo.
          4    : salir
Tu eleccion ? 2
/home/guess
          1    : Imprimir la fecha.
          2, 3 : Imprimir el directorio de trabajo.
          4    : salir
Tu eleccion ? 3
/home/guess
          1    : Imprimir la fecha.
          2, 3 : Imprimir el directorio de trabajo.
          4    : salir
Tu eleccion ? 4
115 %
```

Capítulo 3

Linux para Programadores

Motivación

3.1 Motivación

Prácticamente todo el código ejecutable escrito en el sistema UNIX está desarrollado en C. Para ello se utiliza un compilador de C estándar y esta es probablemente una de las principales razones por las que UNIX es uno de los sistemas operativos más portables en la actualidad. Aplicaciones tales como *Oracle*, la librería *X Window* y el interface gráfico de usuario *Motif* están totalmente escritas en C, por lo que su implantación en una nueva máquina, con un nuevo procesador, y con una arquitectura diferente depende únicamente de que posea un compilador de lenguaje C adecuado. Además otros lenguajes de programación son soportados tales como C++, SmallTalk, Ensamblador, COBOL, FORTRAN, Pascal, ADA, Verilog (un lenguaje de simulación de hardware). El lenguaje de programación C

3.2 El lenguaje de programación C

A pesar de la magnífica portabilidad que poseen los programa construidos en C, existe una pequeña cuestión a tener en cuenta: el estándar que sigue el compilador que estamos utilizando. El primer compilador fue desarrollado por Brian W. Kernighan y Dennis M. Ritchie y su sintaxis se describe adecuadamente en el libro **El Lenguaje de Programación C** de dichos autores. Ellos crearon el estándar **K&R**, el más antiguo. Posteriormente apareció en estándar **ANSI**, que es un super-conjunto de instrucciones del R&R y que posee algunas características que facilitan su uso al programador. Todos los compiladores que alcanzan el estándar ANSI, compilan además programas escritos según el estándar K&R, por lo que si no deseamos comprometer la portabilidad de nuestros programas, deberemos desarrollar según el K&R. Sin embargo esto tiene ciertas desventajas (no existe comprobación ni de número de tipo de los parámetros que se pasan a las funciones). Actualmente lo más conveniente es escribir siguiendo el estándar ANSI por estár ampliamente difundido. Por ejemplo, en OSF/1 y en AIX, los compiladores `cc` (**C** **C**ompiler) son ANSI, y en aquellos sistemas en los que no está disponible el `cc` en su versión ANSI, existe la posibilidad de usar `gcc` que es un compilador de GNU, de dominio público y que está ampliamente portado a la mayoría de las arquitecturas, tal y como ocurre en Linux (incluso está disponible para MS-DOS). En este podemos llamar al compilador de C mediante `cc` debido a que existe un enlace simbólico con este nombre a `gcc` . Programas con un solo módulo

3.3 Programas con un solo módulo

Probablemente, dependiendo de la complejidad del problema que deseamos resolver mediante nuestro programa en lenguaje C, desarrollaremos uno o varios programas fuentes. En esta sección se trata el caso sencillo, en el que únicamente necesitamos crear un solo programa fuente. Por supuesto, necesitamos saber algo de C para desarrollar un programa mínimo. En nuestro caso vamos a construir un programa que invierte los caracteres de cadenas. El fuente que puede realizar esto es:

```
/*
   reverse.c
   Invierte el contenido de una cadena de caracteres.
*/
#include <stdio.h>      /* Puesto que utilizamos NULL */
#include <string.h>     /* Para el uso de strlen() */
void Reverse(                         /* Función que invierte una cadena */
        char *,                    /* Cadena de entrada no invertida */
        char *);                   /* Cadena de salida */
main()                                /* Cuerpo principal del programa */
{
            char str[100];         /* Buffer para realizar inversiones de cadenas */
            Reverse("Hola",str);   /* Invertimos la cadena 'Hola'*/

            /*
               Presentamos en pantalla a través de la salida entandar el
               resultado de la inversión de la cadena 'Hola'
            */
            printf("Reverse (\"Hola\")=%s\n",str);

void Reverse(char *in, char *out)
{
            int i,j;               /* Un par de iteradores */
            int len=strlen(in);    /* Longitud de la cadena a invertir */
            /* Invertimos la cadena */
            for(j=len-1,i=0;j>=0;j--,i++) out[i]=in[j];
            /* NULL es el fin de cadena en C */
            out[len]=(char)NULL;
}
```

Una vez creado con el nombre `reverse.c`, podemos compilarlo y ejecutarlo:

```
dali:~% gcc reverse.c -o reverse
dali:~% reverse
Reverse ("Hola")=aloH
dali:~%
```

Programas multimódulo

3.4 Programas multi-módulo

Es interesante poder reutilizar código ya desarrollado en nuevos programas para crear una biblioteca de funciones. Por ejemplo, sería interesante disponer de una versión testeada y compilada de nuestra función de inversión de cadenas y poderla utilizar cuando nos sea necesario, pero sin necesidad de incluir el código fuente de `Reverse()`. De esta forma nos ahorraremos espacio de disco y tiempo de compilación. La solución es trabajar con programas C multi-módulo. En estos, únicamente un fichero fuente contiene la función `main()` que es la función por la que comienzan a ejecutarse el resto de funciones que hemos usado en nuestro programa. Todas aquellas funciones que formen parte de la biblioteca deben de escribirse en ficheros separados del fichero que contenga el `main()`. Un fichero puede contener más de una función de biblioteca. En nuestro ejemplo, vamos a crear el fichero de biblioteca llamado `Reverse.c` que contendrá la función `Reverse()`. Este será compilado creándose un fichero `Reverse.o` que contiene código máquina pero que no es un fichero ejecutable. Este debe de ser link-ado junto con la compilación del fichero `reverse.c` que contiene el `main()`. Existe además un fichero llamado `Reverse.h` que debe de contener aquellas definiciones de funciones (prototipos de funciones) tipos y estructuras de datos que deben de ser conocidos para el uso de las bibliotecas. En nuestro ejemplo, se generan los siguientes ficheros:

1. El fichero `Reverse.h`:

```
/*
        Reverse.h
        Fichero cabecera con el prototipo de la función Reverse().
*/
void reverse(           /* Invierte */
        char *,         /* la cadena de entrada */
        char *);        /* la cadena de salida */
```

2. El fichero `Reverse.c`:

```
/*
        Reverse.c
        Fichero que contiene el código que de la función Reverse().
*/
#include <stdio.h>      /* Puesto que utilizamos NULL */
#include <string.h>     /* Para el uso de strlen() */
#include "Reverse.h"    /* La definición de la biblioteca (a efectos de
                           chequeo de tipos en la compilación) */
void Reverse(char *in, char *out)
{
        int i,j;                /* Un par de iteradores */
        int len=strlen(in);     /* Longitud de la cadena a invertir */
        /* Invertimos la cadena */
        for(j=len-1,i=0;j>=0;j--,i++) out[i]=in[j];
        /* NULL es el fin de cadena en C */
        out[len]=NULL;
}
```

3. Y el fichero en el que usamos la biblioteca construida:

```
/*
        reverse.c
        Testea el uso de la función Reverse.
*/
#include <stdio.h>      /* Usamos el printf() */
#include "Reverse.h"    /* Prototipo de Reverse(). Necesario ! */
main()                          /* Cuerpo principal del programa */
{
        char str[100];          /* Buffer para realizar inversiones de cadenas */
        Reverse("Hola",str);    /* Invertimos la cadena 'Hola'*/
        /*
           Presentamos en pantalla a través de la salida entandar el
           resultado de la inversión de la cadena 'Hola'
        */
        printf("Reverse (\"Hola\")=%s\n",str);
}
```

Para compilar nuestro programa deberemos de ejecutar:

```
dali:~% gcc -c Reverse.c
dali:~% ls -l Reverse.o
-rw-r--r--   1 vi       wheel         242 May 11 15:32 Reverse.o
dali:~% gcc reverse.c Reverse.o -o reverse
```

Y para ejecutar el ejemplo:

```
dali:~% reverse
Reverse ("Hola")=aloH
dali:~%
```

El sistema de dependencia entre ficheros: make

3.5 El sistema de dependencia entre ficheros: make

El trabajar con programas multi-módulo tiene una desventaja: cuando nos damos cuenta, tenemos muchos ficheros que contienen funciones de biblioteca y no recordamos que dependencias existen entre ellos. Por ejemplo, supongamos que retocamos un fichero de biblioteca y que lo recompilamos, pero de este fichero dependen una larga lista de ficheros también de biblioteca. Deberemos recompilar cada uno de ellos si deseamos que los cambios realizados se noten en el programa que estamos desarrollando. Para solventar este problema, UNIX tiene la utilidad `make` que recompilará aquellos ficheros que necesiten ser recompilados, tras una modificación. La sintaxis de este comando es:

`make -f` *makefile*

El fichero *makefile* es un fichero texto que contiene las dependencias entre ficheros necesarias para que `make` trabaje. Por defecto, `make` buscará un fichero llamado `makefile` o `Makefile` en el directorio de trabajo. Si el nombre de fichero de dependencias no es éste, o está en otro directorio deberemos indicar su nombre a través del flag `-f` . La forma general de un fichero `makefile` es:

```
OBJETIVO: lista de dependencias con otros objetivos anteriores
          lista de comandos a realizar par conseguir OBJETIVO
```

Por ejemplo, un `makefile` para nuestro programa `reverse` podría ser:

```
#
# makefile para 'reverse'
#
#
# El fichero 'Reverse.o' depende de los fichero 'Reverse.c' y 'Reverse.h'.
# Si uno de ellos es modificado, entonces se ejecuta el comando:
#           cc -c Reverse.c -o Reverse.o
# que compila la función y crea un fichero llamado 'Reverse.o'
#
Reverse.o:  Reverse.c Reverse.h
               cc -c Reverse.c -o Reverse.o
#
# El fichero ejecutable 'reverse' sera compilado si cambia el fichero
# 'Reverse.o' o 'reverse.c'.
#
reverse:    Reverse.o reverse.c
               cc reverse.c Reverse.o -o reverse
```

Para compilar el programa `reverse` escribiremos:

```
dali:~% make reverse
cc -c Reverse.c -o Reverse.o
cc reverse.c Reverse.o -o reverse
dali:~%
```

Debemos de dar el parámetro `reverse` porque este es el objetivo final que va a conseguir la utilidad `make` . Si deseamos que únicamente se genere `Reverse.o`, debiéramos de escribir:

```
dali:~% make Reverse.o
cc -c Reverse.c -o Reverse.o
dali:~%
```

No es necesario escribir `make -f makefile reverse` porque `make` busca por defecto el fichero `makefile`. La utilidad `make` puede usarse en general para establecer dependencias entre ficheros de cualquier clase, por ejemplo, para ficheros en formato TEX. Por último, cuando intentamos compilar y no hemos realizado ninguna modificación, `make` no trabaja y nos presenta el siguiente mensaje:

```
dali:~% !!
make reverse
make: 'reverse' is up to date.
dali:~%
```

El debugger de UNIX: dbx

3.6 El debugger de UNIX: dbx

Cuando ejecutamos un programa escrito por nosotros y tenemos errores en tiempo de ejecución usaremos un debugger para encontrarlos. Normalmente estos errores provocan que la memoria que ocupa el programa que aborta sea copiada en el disco creando un fichero `core`. Este fichero es usado por el debugger para averiguar en que línea del código fuente se ha producido el error. En Linux el debugger se llama `gdb` (**G**enius **D**e**B**ugger). La forma de invocarlo es:

`gdb` *fichero_ejecutable*

Lo primero que debemos hacer es compilar nuestro programa para ser depurado. Esto se realiza con la opción `-g` del compilador de C. Una forma rápida de realizar esto es modificando el fichero `makefile`, para insertar el flag de compilación `-g` con lo que obtenemos:

```
dali:~% make reverse
cc -g -c Reverse.c -o Reverse.o
cc -g reverse.c Reverse.o -o reverse
dali:~%
```

Para invocar el debugger:

```
dali:~% gdb reverse
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i486-slackware-linux),
Copyright 1994 Free Software Foundation, Inc...
(gdb)
```

Para ejecutar el programa (sin paradas provocadas por nosotros):

```
(gdb) run
Starting program: /home/vi/reverse
Reverse ("Hola")=aloH
Program exited with code 026.
(gdb)
```

De esta forma, si lo que tenemos es una parada en tiempo de ejecución, el debugger se para en la línea que causa el problema. Si lo que deseamos es establecer puntos de ruptura (breakpoints):

```
(gdb) list (Listamos el programa que estamos depurando)
3               Testea el uso de la función Reverse.
4       */
5
6       #include <stdio.h>      /* Usamos el printf() */
7       #include "Reverse.h"    /* Prototipo de Reverse(). Necesario ! */
8
9       main()                          /* Cuerpo principal del programa */
10      {
11              char str[100];          /* Buffer para realizar inversiones de cadenas */
12              Reverse("Hola",str);    /* Invertimos la cadena 'Hola'*/
(gdb) break 12 (establecemos un punto de ruptura en la linea 12)
Breakpoint 1 at 0x109f: file reverse.c, line 12.
(gdb) run (ejecutamos el programa)
Starting program: /home/vi/reverse
Breakpoint 1, main () at reverse.c:12
12              Reverse("Hola",str);    /* Invertimos la cadena 'Hola'*/
(gdb)
```

Para trazar la ejecución del programa paso a paso realizamos:

```
(gdb) step
Reverse (in=0x1074 "Hola", out=0xbffffc2c "tically linked\n") at Reverse.c:14
14              int len=strlen(in);     /* Longitud de la cadena a invertir */
(gdb) step
strlen (str=0x1074 "Hola") at strlen.c:29
(gdb) step
Reverse (in=0x1074 "Hola", out=0xbffffc2c "tically linked\n") at Reverse.c:17
17              for(j=len-1,i=0;j>=0;j--,i++) out[i]=in[j];
(gdb) step
20              out[len]=(char)NULL;
(gdb) print out[1]
$1 = 108 'l'
(gdb) print out[0]
$2 = 97 'a'
(gdb) print *out
$3 = 97 'a'
(gdb)
```

Y como vemos es posible acceder a las variables del programa cuando lo estamos ejecutando paso a paso. `gdb` dispone de otros muchos comandos que se escapan del interés de este curso. Para obtener información acerca de ellos podemos escribir `help` en la línea de comandos. Para salir del debugger teclearemos el comando `quit` .