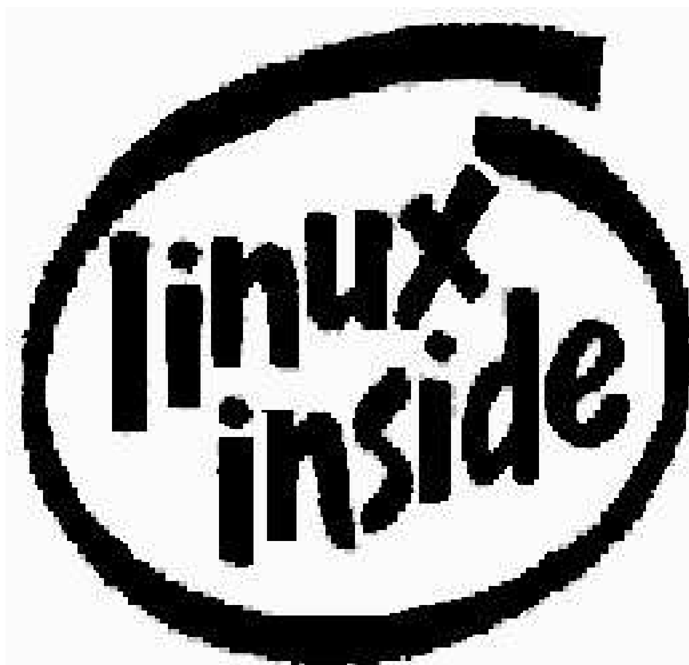
INTRODUCCIÓN AL SISTEMA OPERATIVO *UNIX*

Antonio Villalón Huerta <`toni@shutdown.es`>

Febrero, 2008



'The bad reputation UNIX has gotten is totally undeserved, laid on by people who don't understand, who have not gotten in there and tried anything.'

Jim Joyce

This is 100 % free of Micro$oft products. By LaTeX for Linux.

Índice

1. INTRODUCCIÓN **5**
1.1. ¿Qué es un Sistema Operativo? 5
1.2. Historia del Sistema Unix 5
1.3. Características de Unix 6
1.4. Linux 8
1.5. Nociones básicas 9

2. LA PRIMERA SESIÓN **10**
2.1. Nociones previas: *login* y *password* 10
2.2. Cambio de claves: la orden `passwd` 11
2.3. Los archivos `/etc/profile` y `.profile` 11
2.4. Fin de una sesión: `exit` y `logout` 12

3. ÓRDENES BÁSICAS DE UNIX **14**
3.1. Introducción 14
3.2. Gestión de archivos y directorios 14
3.2.1. `ls` 14
3.2.2. `cd` 15
3.2.3. `pwd` 15
3.2.4. `mkdir` 15
3.2.5. `touch` 15
3.2.6. `cp` 16
3.2.7. `mv` 16
3.2.8. `rm` 17
3.2.9. `rmdir` 17
3.2.10. `ln` 17
3.2.11. `chmod` 18
3.3. Ayuda en línea: `man` 19
3.4. Información del sistema 20
3.4.1. `date` 20
3.4.2. `uname` 20
3.4.3. `id` 20
3.4.4. `who` 21
3.4.5. `w` 21
3.4.6. `last` 21
3.4.7. `ps` 21
3.4.8. `kill` 22
3.4.9. `du` 22
3.4.10. `df` 22
3.5. Tratamiento básico de archivos 23
3.5.1. `file` 23
3.5.2. `cat` 23
3.5.3. `more/less` 24
3.6. Tratamiento avanzado de archivos 24
3.6.1. `head` 24
3.6.2. `tail` 24
3.6.3. `cmp` 24
3.6.4. `diff` 25
3.6.5. `grep` 25
3.6.6. `wc` 26
3.6.7. `sort` 26
3.6.8. `spell` 26

3.7. Órdenes de búsqueda . . . 27
3.7.1. `find` . . . 27
3.7.2. `which` . . . 28
3.8. Compresión y empaquetado . . . 28
3.8.1. `gzip` . . . 28
3.8.2. `tar` . . . 28
3.8.3. `bzip2` . . . 30
3.9. Otras órdenes . . . 30
3.9.1. `echo` . . . 30
3.9.2. `cal` . . . 30
3.9.3. `passwd` . . . 31
3.9.4. `clear` . . . 31
3.9.5. `sleep` . . . 31
3.9.6. `nohup` . . . 31
3.9.7. `alias` . . . 31

4. COMUNICACIÓN ENTRE USUARIOS **33**
4.1. Introducción . . . 33
4.2. La orden `write` . . . 33
4.3. La orden `talk` . . . 33
4.4. La orden `mail` . . . 34

5. EL EDITOR DE TEXTOS `vi` **36**
5.1. Introducción . . . 36
5.2. Comenzando con `vi` . . . 36
5.3. Saliendo del editor . . . 37
5.4. Tratamiento del texto . . . 37
5.5. Otras órdenes de `vi` . . . 38
5.6. Órdenes orientadas a líneas . . . 39

6. UNIX Y REDES: APLICACIONES **41**
6.1. Introducción . . . 41
6.2. `tin` . . . 41
6.3. Uso de `tin` . . . 41
6.4. `lynx` . . . 42
6.5. Uso de `lynx` . . . 43
6.6. `gopher` . . . 43
6.7. `ftp` . . . 44
6.8. `telnet` . . . 46
6.9. `finger` . . . 46
6.10. `elm` . . . 47
6.11. `pine` . . . 49

7. CONCEPTOS DEL SISTEMA OPERATIVO UNIX **50**
7.1. Ficheros . . . 50
7.2. Permisos de los archivos . . . 51
7.3. Archivos ejecutables, imágenes y procesos . . . 52
7.4. El *shell* . . . 52
7.5. Programación en *shell* . . . 53
7.6. Organización de directorios . . . 54
7.7. Planos de trabajo . . . 55
7.8. Entrada y salida . . . 56

8. SEGURIDAD BÁSICA DEL USUARIO **59**
8.1. Sistemas de contraseñas 59
8.2. Archivos *setuidados* y *setgidados* 60
8.3. Privilegios de usuario 61
8.4. Cifrado de datos 61
8.5. Bloqueo de terminales 62

1. INTRODUCCIÓN

1.1. ¿Qué es un Sistema Operativo?

En un entorno informático existen cuatro grandes componentes: el usuario, las aplicaciones, el sistema operativo y el *hardware*. El **usuario** es la persona que trabaja con el ordenador, ejecutando **aplicaciones** (programas) para llevar a cabo ciertas tareas: editar textos, diseñar, navegar por Internet... El **hardware** es la parte del entorno que podemos romper con un martillo: la memoria, el disco duro, el lector de CD–ROMs, la placa base... ¿Y el sistema operativo? El **sistema operativo** es simplemente un trozo de código (un programa, aunque no una aplicación) que actúa como intermediario entre el usuario y sus aplicaciones y el *hardware* de un computador, tal y como se muestra en la figura 1.

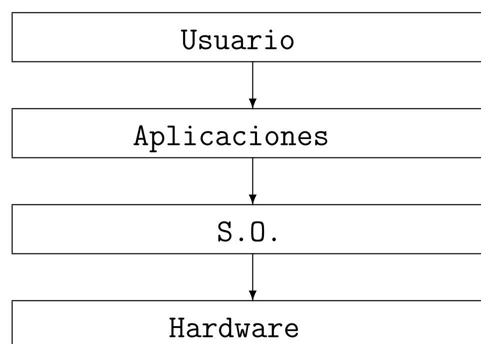


Figura 1: Capas de un entorno informático.

El propósito de un sistema operativo (a partir de ahora lo llamaremos S.O) es proporcionar un entorno en el cual el usuario pueda ejecutar programas de forma cómoda y eficiente. Un S.O. no lleva a cabo ninguna función útil por sí mismo, sino que sólo se limita a proporcionar un entorno donde los programas puedan desarrollar sus trabajos útiles de cara al usuario; el S.O. controla la ejecución de tales programas para poder prevenir los errores y el uso inadecuado del ordenador: por este motivo, se ha de estar ejecutando permanentemente en la computadora (al menos el núcleo o *kernel* del S.O).

Ejemplos de SS.OO. son OS/390, Unix, Windows NT, Windows 95/98/2000... (Windows 3.1 no es un S.O., sólo un entorno gráfico), OS/2 o VMS. Cada uno suele aplicarse en un cierto campo y, aunque ninguno es perfecto, las diferencias de calidad entre ellos pueden ser abismales, sobre todo si hablamos de los 'operativos de sobremesa', como Windows 9∗, frente a sistemas serios.

1.2. Historia del Sistema Unix

En 1964 diversos organismos estadounidenses (entre ellos el MIT, *Massachusetts Institute of Technology*) se propusieron diseñar un ambicioso sistema operativo denominado MULTICS (*Multiplexed Information and Computing System*), capaz de proporcionar una gran potencia de cómputo y de almacenar y compartir grandes cantidades de información. Como este proyecto era demasiado ambicioso para la época, los trabajos sobre MULTICS fracasaron; sin embargo, las ideas usadas en el desarrollo del sistema operativo sirvieron de base para el diseño de un nuevo sistema al que irónicamente se denominaría UNICS (*'Uniplexed' Information and Computing System*), nombre que posteriormente desembocó en Unix.

Durante 1969, en los laboratorios Bell, Ken Thompson comenzó el diseño del sistema Unix sobre un viejo DEC PDP–7 apartado del uso. Poco más tarde se le unieron otros científicos, como Dennis Ritchie. Juntos escribieron un sistema de tiempo compartido (multitarea) de uso general,

lo bastante cómodo y eficiente como para trasladarlo a una máquina más potente (un PDP-11-20, en 1970). Tres años más tarde, Thompson y Ritchie reescribieron el núcleo del sistema operativo en C (anteriormente lo estaba en lenguaje ensamblador), lo cual dió a Unix su forma esencial, tal y como lo conocemos hoy en día.

Poco después de tener el núcleo del sistema escrito en un lenguaje de alto nivel, Unix fue introduciéndose rápidamente en las universidades, con fines educacionales, hasta llegar a su uso comercial en laboratorios, centros de procesamiento de datos, centros de operaciones en compañías telefónicas estadounidenses, etc. La disponibilidad pública del código fuente del sistema, su portabilidad y su potencia fueron factores claves en esta rápida expansión de Unix.

En estos primeros tiempos de Unix (Versión 6), el código fuente del sistema era fácil de conseguir. Sin embargo, a partir de la Versión 7, el auténtico 'abuelo' de los sistemas Unix modernos, AT&T se comenzó a dar cuenta que tenía entre manos un producto comercial rentable, por lo que se prohibió el estudio del código fuente para no hacer peligrar la versión comercial.

A finales de 1978, después de distribuir la Versión 7, la responsabilidad y el control administrativo de Unix pasó a manos de USG (*Unix Support Group*), integrado en AT&T, que sacó al mercado Unix System III (1982) y posteriormente el famoso System IV (1983), hasta llegar a Unix System V Release 4 (SVR4), uno de los sistemas más extendidos en la actualidad, y sus posteriores modificaciones.

Sin embargo, aunque AT&T deseaba tener el control absoluto sobre las distribuciones de Unix, debido a la potencia de este sistema, muchos grupos de desarrollo ajenos a la compañía han trabajado en la mejora de Unix. Uno de estos grupos, posiblemente el que más influencia ha tenido, ha sido la Universidad de California en Berkeley, que desarrolló el sistema Unix hasta alcanzar 3BSD (*Berkeley Software Distribution*), a principios de los 80 (tomado como estándar en el Departamento de Defensa estadounidense y su red DARPANet), y que hoy sigue desarrollando software como Unix 4BSD en sus diferentes versiones.

Otras compañías que defienden o han defendido sus propias versiones de Unix pueden ser Microsoft (Xenix), DEC (Ultrix), IBM (AIX) o Sun Microsystems (Solaris). Cada una de ellas desarrolla diferentes sistemas; sin embargo, las características básicas son comunes en todos, por lo que conociendo cualquier tipo de Unix mínimamente no es difícil poder dominar otro tipo en poco tiempo. En la tabla 1 se presenta un resumen de los principales clones de Unix y la compañía que está detrás de cada uno de ellos.

1.3. Características de Unix

Podemos considerar a Unix como el sistema operativo de propósito general más potente, flexible y robusto existente hoy en día; son muchísimas las ventajas que presenta el uso de cualquier clon de Unix (de los comentados anteriormente) frente a otros entornos más habituales como MS–DOS o Windows NT, pero una de las más visibles para el usuario que se inicia en el mundo Unix es la capacidad de trabajar en un simple PC casi de la misma forma que lo haría en un superservidor de gama alta (guardando las distancias oportunas, por supuesto). Entre las características más destacables de todos los sistemas Unix podemos citar las siguientes:

- Multitarea.
  Por multitarea entendemos la capacidad de ejecutar varios procesos (para nosotros, varios programas) sin detener la ejecución de cada uno de ellos de una forma perceptible para el usuario; aunque en cada momento, por cuestiones de arquitectura, en un procesador se está ejecutando – al menos efectivamente – un único programa, el sistema operativo es capaz de alternar la ejecución de varios de una forma tan rápida que el usuario tiene una sensación de dedicación exclusiva. La multitarea de Unix está basada en prioridades, y es el núcleo

Nombre	**Compañía u organización**
386BSD	Gratis en Internet
AIX	IBM
Artix	Action Inst.
A/UX	Apple
BSD	University of California at Berkeley
BSD-Lite	University of California at Berkeley
BSD/386	Berkeley Software Design (BSDI)
Coherent	Mark Williams Company
Dynix	Sequent
FreeBSD	Gratis en Internet
HP-UX	Hewlett-Packard
Hurd (GNU)	Free Software Foundation
IC-DOS	Action Inst.
Interactive Unix	Sun Microsystems
Irix	Silicon Graphics
Linux	Gratis en Internet
Lynx OS	Lynx RT Systems
Mach	Carnegie-Mellon University
Minix	Distribuido con el libro de Tanenbaum (Bibliografía)
MKS Toolkit	Mortice Kern Systems
NetBSD	Gratis en Internet
Nextstep	Next
OSF/1	Digital Equipment Corporation (DEC)
PC Xenix	IBM
QNX	QNX Software Systems
RTX/386	VenturCom
SCO Unix	Santa Cruz Operation
Solaris	Sun Microsystems
SunOS	Sun Microsystems
System V Unix	Varias versiones para PC
Ultrix	Digital Equipment Corporation (DEC)
Unicos	Cray Research
Unixware	Novell
V/386	Microport
V/AT	Microport
Xenix	Microsoft

Cuadro 1: *Flavours* de Unix

del sistema operativo el encargado de otorgar el control de los recursos a cada programa en ejecución: un proceso se ejecuta hasta que el sistema operativo le cede el paso a otro.

- Multiusuario.
  La capacidad multiusuario de Unix permite a más de un usuario ejecutar simultáneamente una o varias aplicaciones (incluso la misma) sobre una misma máquina; esto es posible gracias a la capacidad de multitarea del operativo, que es capaz de repartir el tiempo de procesador entre diferentes procesos en ejecución. Unix gestiona y controla a sus usuarios, determinando quién y cuándo puede conectarse al sistema y, una vez dentro del mismo, qué puede hacer y a qué recursos tiene acceso.

- *Shell* programable.
  El *shell* o intérprete de órdenes de un sistema operativo es el proceso que sirve de interfaz entre el usuario y el núcleo (*kernel*) del operativo; en el caso de Unix, los diferentes *shells* existentes (hablaremos de ellos más adelante) son programables, ya que 'entienden' un lenguaje denominado *shellscript* y que se conforma con todas las órdenes del operativo junto a estructuras de control, como los bucles. La programación en *shellscript* puede resultar muy compleja y permite un elevado control del sistema, facilitando por ejemplo la automatización de tareas costosas, como las copias de seguridad.

- Independencia de dispositivos.
  El sistema operativo Unix trata a cualquier elemento *hardware* de la máquina como un sencillo fichero, de forma que para acceder a un dispositivo el usuario sólo ha de conocer el nombre del archivo que lo representa. Este acceso es independiente del tipo y modelo del dispositivo, y se realiza mediante las primitivas de acceso a ficheros habituales (`open()`, `seek()`, `read()`, `write()`...). Por ejemplo, para acceder a un módem, las aplicaciones – y por tanto, los usuarios finales – utilizarán estas primitivas independientemente de la marca y modelo del mismo, de forma que si en un momento dado el *hardware* cambia (se compra un módem mejor) el administrador se encargará de enlazar un nuevo controlador en el núcleo, y a partir de ese momento el cambio será completamente transparente al usuario y sus aplicaciones.

1.4. Linux

Aunque este curso está orientado a Unix en general, hemos de elegir uno de los muchos clones del operativo para trabajar durante las clases; y como la mayor parte de usuarios domésticos de Unix opta por Linux, hemos elegido este entorno. Linux no es sino uno más de los muchos sistemas Unix que actualmente están en la calle; en su origen, el núcleo del sistema fue desarrollado por un estudiante finlandés llamado Linus Torvalds (de ahí el nombre del sistema, *Linus´ Unix*) como su Proyecto Final de Carrera. Su objetivo era crear un sustituto mejorado de Minix, otro clon de Unix para PC creado por Andrew Tanenbaum con el objetivo de servir de base en el estudio de sistemas operativos. Programadores de todo el mundo han contribuido posteriormente a mejorar Linux con aplicaciones y mejoras contínuas, por lo que se puede decir que este sistema operativo es hoy en día el resultado del trabajo de miles de usuarios.

Linux, considerado por muchos como el mejor sistema operativo para los procesadores de la familia del i386, contiene todas las características que presenta un Unix normal: multitarea, protocolos para trabajo en redes, memoria virtual, independencia de dispositivos, etc. Puede trabajar con cualquier procesador 386 o superior (no funciona en 286), con unos requerimientos mínimos de 10 Mb. de disco duro y 2 Mb. de memoria RAM.

Linux se presenta al usuario bajo lo que se llaman **distribuciones**: un núcleo Linux rodeado de diferentes aplicaciones y con un determinado aspecto, todo ello empaquetado conjuntamente y listo para ser instalado. Algunas de las distribuciones más conocidas son Debian, Slackware, Red Hat, SuSe o Mandrake, cada una de las cuales tiene puntos a favor y puntos en contra y es susceptible de ser utilizada en un determinado entorno (aunque realmente, cualquier distribución de Linux puede

realizar las mismas tareas que cualquier otra).

Actualmente es muy fácil encontrar cualquier información que necesitemos de Linux utilizando INet, tanto en multitud de páginas *web* como en grupos de noticias como `comp.os.linux.*` o `es.comp.linux.*`; en estos *newsgroups* podemos encontrar respuesta a todas nuestras dudas sobre el sistema. También tenemos a nuestra disposición numerosa bibliografía sobre el operativo, desde su uso básico a su administración más avanzada.

1.5. Nociones básicas

El curso se va a desarrollar en una máquina Unix (Linux en concreto) a la que vamos a conectar mediante algún protocolo de terminal remota, como `telnet` o SSH; a este tipo de equipos, a los que conectamos a través de la red, se les denomina *servidores*[1]. Cada servidor suele tener un nombre nombre y una dirección IP asociados: por ejemplo el nombre puede ser similar a `andercheran.aiind.upv.es`, y la dirección algo como `158.42.22.41`.

El hecho de trabajar en una máquina remota implica que todo lo que nosotros hagamos lo haremos en tal ordenador, no en el que tenemos delante y que usamos simplemente para conectar al servidor. Por decirlo de una forma sencilla, de este sólo aprovechamos el teclado y el monitor: por tanto, si guardamos un archivo en el disco duro, este archivo estará en el disco del servidor, no en el del ordenador que tenemos delante. Así, en cuestión de velocidad, rendimiento, espacio en disco. . . no dependerá para nada el que estemos conectados desde una u otra terminal: tan rápido (o tan lento) podremos trabajar si conectamos desde un 286 como si lo hacemos desde un Pentium. Sólo dependerá de la carga que esté soportando el servidor en esos momentos.

Si trabajamos sobre un PC con Windows, para poder acceder al servidor Linux una vez hayamos arrancado el ordenador local debemos teclear desde MS-DOS:

```
c:\> telnet andercheran.aiind.upv.es
```

Si trabajamos sobre otra máquina Unix conectaremos de la misma forma:

```
rosita:~$ telnet andercheran.aiind.upv.es
```

De esta forma podremos conectar con el servidor y comenzar a trabajar, tal y como se explica en el Capítulo 2.

[1]Realmente, el término 'servidor' es mucho más amplio, pero para nosotros será simplemente, al menos de momento, el sistema Unix remoto.

2. LA PRIMERA SESIÓN

2.1. Nociones previas: *login* y *password*

Como ya hemos comentado en el capítulo anterior, Unix es un sistema operativo **multiusuario**; en todos los entornos que poseen esta capacidad debe existir alguna manera de que el sistema mantenga un control de sus usuarios y las peticiones que realizan: esto se consigue mediante lo que se conoce como **sesión**, que no es más que el proceso existente desde que el usuario se identifica y autentica ante el sistema hasta que sale del mismo.

Para trabajar en un sistema Unix cada usuario debe iniciar una sesión; para ello es necesario que el usuario tenga asociado un nombre de acceso al sistema (conocido y otorgado por el administrador) denominado *login*, y una contraseña (privada y sólo conocida por el propio usuario) denominada *password*. Así, cada vez que conectemos a una máquina Unix, ya sea mediante una terminal serie o mediante una terminal remota (por ejemplo utilizando `telnet` o `ssh`), lo primero que el sistema va a hacer es pedirnos nuestro *login* y nuestro *password*, para verificar que realmente somos nosotros los que nos disponemos a iniciar la sesión y no otra persona. Esta clave personal, que trataremos más adelante, no aparecerá en pantalla al escribirla, por cuestiones obvias de seguridad (también por motivos obvios, no se escribirán asteriscos ni nada similar, a diferencia de lo que sucede en los mecanismos de autenticación de otros sistemas operativos).

Tras haber verificado nuestra identidad, se nos permitirá el acceso al sistema; después de darnos una cierta información, como la hora y el lugar desde donde nos conectamos la última vez o si tenemos o no correo electrónico, la máquina quedará a la espera de que nosotros empecemos a trabajar introduciéndole órdenes. Para ello, nos mostrará el *prompt* de nuestro intérprete de órdenes, también conocido como símbolo del sistema, y ya estaremos listos para comenzar a introducir instrucciones.

En la siguiente figura vemos un ejemplo real de todos estos detalles:

```
          #####    ####    ####      #     #####    ##
          #    #  #    #  #          #       #     #  #
          #    #  #    #   ####      #       #    #    #
          #####   #    #       #     #       #    ######
          #   #   #    #  #    #     #       #    #    #
          #    #   ####    ####      #       #    #    #

rosita login: toni                     (Nuestro nombre de usuario)
Password:                              (Nuestra clave, que no
                                       aparece en pantalla)
Last login: Fri Oct  4 01:01:27 on tty1  (Ultima conexion)
Linux 1.3.50                           (Version del S.O.)
You have mail.                         (Tenemos correo?)
rosita:~$                              (Ahora podemos comenzar a
                                       introducir ordenes al computador)
```

Como hemos comentado, Unix nos indica que está en disposición de recibir órdenes mostrándonos en pantalla el *prompt* o símbolo del sistema; este depende del *shell* con el que estemos trabajando y además es algo que podemos modificar a nuestro gusto, pero habitualmente este *prompt* viene indicado por los símbolos `'$'`, `'%'` o `'>'`, solos o precedidos de información como el nombre del *host* al que hemos conectado, el *login* de entrada al mismo, o el directorio en el que estamos trabajando en cada momento. Un ejemplo habitual de *prompt* es el siguiente:

```
rosita:~$
```

Una vez hemos accedido al sistema, parece obvio que nos encontramos en un determinado directorio del mismo; este directorio de entrada es definido por el administrador de la máquina para cada uno de sus usuarios, y se almacena en la variable de entorno *$HOME*, por lo que nos referiremos habitualmente a él como 'el directorio *$HOME*'. Generalmente varios o todos los usuarios comparten el padre de sus directorios de entrada (es decir, todos 'cuelgan' del mismo sitio), y este suele ser de la forma `/home/cunixXX/`, `/export/home/cunixXX/` o similar, aunque nada impide que el administrador elija otros directorios para sus usuarios. Podemos comprobar cual es nuestro *$HOME* mediante la orden `pwd`, nada más acceder a la máquina:

```
rosita:~$ pwd
/home/toni
rosita:~$
```

2.2. Cambio de claves: la orden `passwd`

Durante la primera sesión, cuando ya hemos entrado dentro del sistema Unix, lo primero que hemos de hacer por motivos de seguridad es cambiar nuestra contraseña. Ésta, por norma general, bien no existe (en cuyo caso no se nos habría preguntado por el *password*), o bien coincide con nuestro nombre de usuario, el *login*. Hemos de poner una nueva contraseña a nuestra cuenta en el sistema, utilizando para ello la orden `passwd`:

```
rosita:~$ passwd
Changing password for toni
Old password:
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
Re-enter new password:
Password changed.
rosita:~$
```

Aunque trataremos el tema de la seguridad más adelante, hemos de saber que si alguien es capaz de adivinar nuestra clave podrá tener acceso a nuestros documentos, trabajos, cartas, etc., podrá modificarlos, borrarlos, o simplemente leerlos. Como no deseamos que esto ocurra, hemos de utilizar una clave que no sea una palabra común (¡mucho menos nuestro nombre de usuario!); una combinación de letras mayúsculas y minúsculas, y algún número, sería ideal. Hemos de huir de claves como *patata*, *internet*, o *beatles*. Podríamos utilizar una clave como *99Heyz0*, *cu43Co3* o *W34diG*. Sin embargo, esta clave la hemos de recordar para poder entrar de nuevo al sistema. No conviene apuntarla en un papel; es mejor acordarse de ella de memoria, utilizar algo familiar a nosotros que combine letras y números, como nuestras dos primeras iniciales seguidas de tres números del DNI y otra letra mayúscula, o algo parecido. Como hemos dicho, trataremos el tema de la seguridad más adelante, pero es necesario insistir en que la elección de una buena contraseña es fundamental para la integridad y privacidad de nuestros datos...

2.3. Los archivos `/etc/profile` y `.profile`

Cada usuario de un sistema Unix tiene definido un *shell* por defecto, un intérprete de órdenes que será el programa que le va a servir de interfaz para con el sistema operativo; con el *shell* más estándar de Unix, denominado *Bourne Shell* (`sh`), así como con su versión mejorada (*Bourne Again Shell*, `bash`), utilizada por defecto en Linux, tenemos la posibilidad de definir un fichero que se ejecute cada vez que conectemos a la máquina. Dicho archivo se denomina `.profile` y está situado en el directorio *$HOME* de cada uno de nosotros (recordemos, en el directorio en el cual entramos tras teclear el *login* y la contraseña, por ejemplo `/home/cunixXX/`).

Simplemente creando un archivo con este nombre el *shell* tratará de interpretarlo como un proceso por lotes cada vez que entremos en el sistema; no hemos de otorgarle ningún permiso especial

ni definir ningún atributo sobre él, sólo llamarle `.profile`. Así, tendremos un proceso por lotes que se ejecutará cada vez que conectemos a la máquina, de una forma relativamente similar al `autoexec.bat` de MS-DOS; en él podemos introducir todos los mandatos cuyo resultado queremos obtener nada más entrar en nuestra cuenta. Por ejemplo, si siempre que conectemos queremos ver la fecha y la hora en la que nos encontramos, en el archivo `.profile` incluiremos una línea con la instrucción `date`; si además queremos ver quién está conectado en esos momentos en la máquina, incluiremos la orden `who`. También lo podemos utilizar para recordar cosas que tenemos que hacer, mediante la orden `echo` (por ejemplo, podemos teclear una línea con la orden `echo 'HABLAR CON EL ADMINISTRADOR'`): de esta forma, cada vez que entremos en nuestra cuenta nos aparecerá tal mensaje en pantalla...

En Unix es necesario distinguir entre 'conectar al sistema' y 'encender el sistema'; una máquina Unix posiblemente se arranque muy pocas veces (recordemos que estamos ante un sistema operativo preparado para funcionar de forma ininterrumpida durante largos periodos de tiempo, a diferencia de las distintas versiones de Windows), y cuando esto suceda se ejecutarán unos programas que no vamos a ver aquí, ya que su configuración corresponde al administrador de la máquina. Frente al arranque del sistema, los usuarios inician sesión (conectan) en muchas ocasiones, y es en cada una de ellas cuando se ejecutan los ficheros como `.profile`, del que estamos hablando.

En el archivo `.profile` podremos tener tantas intrucciones (de las muchas que veremos posteriormente) como deseemos; sólo hemos de recordar que en una línea sólo hemos de llamar a una orden (no vamos a ver aquí los separadores de órdenes, como '`&&`', '`||`' o '`;`'), y que las líneas que comiencen por el símbolo '`#`' son comentarios, que serán ignorados.

Veamos un posible ejemplo de nuestro archivo `.profile`:

```
rosita:~$ cat .profile
date      # Nos indica dia y hora
echo "Bienvenido de nuevo"   # Muestra este mensaje en pantalla
#who      # Ignorado, ya que la linea comienza por '#'
fortune   # Frase, comentario o idea famosa o graciosa
rosita:~$
```

Otro archivo equivalente al `.profile` es el `/etc/profile`. Su función es análoga, con la diferencia de que en `/etc/profile` se especifican órdenes y variables comunes a todos los usuarios. Como este archivo es administrado por el `root` del sistema, no entraremos en más detalles con él.

2.4. Fin de una sesión: `exit` y `logout`

Existen muchas formas de finalizar una sesión en un sistema Unix, dependiendo del tipo de acceso que se haya realizado: si hemos conectado utilizando un módem, podemos simplemente colgar el teléfono; también podemos pulsar repetidamente `Ctrl-D` hasta matar a nuestro intérprete de órdenes, apagar el ordenador... Sin embargo, estas formas de terminar una sesión no son en absoluto recomendables, tanto por motivos de seguridad como por la estabilidad del sistema. Para finalizar correctamente nuestro trabajo en el ordenador, debemos desconectar con una orden implementada para esta tarea, como pueda ser `exit` o `logout`. Estas instrucciones se encargarán de cerrar completamente nuestra conexión y finalizar nuestros procesos (a no ser que hayamos especificado lo contrario con `nohup`, que veremos más adelante) de tal forma que no comprometamos al sistema completo; esto se realiza, básicamente, enviando una señal SIGKILL sobre nuestro intérprete de órdenes, que es nuestro proceso principal. Si él muere, muere también la conexión establecida.

Utilizando el intérprete de órdenes `bash`, adjudicado por defecto bajo Linux, en el archivo `.bash_logout` tenemos la opción de especificar ciertas tareas que queremos realizar siempre que desconectemos del sistema (sería el contrario al `.profile`, estudiado anteriormente). Si en lugar de `bash` utilizáramos el intérprete *C-Shell*, estas tareas a realizar antes de la desconexión se han de

definir en el archivo `.logout`, mientras que el equivalente al `.profile` en este caso se denominaría `.cshrc`. Ahondaremos sobre estas cuestiones cuando estudiemos los diferentes intérpretes de mandatos disponibles en Unix.

Para finalizar, hay que resaltar que tanto `logout` como `exit` son instrucciones internas del sistema, es decir, no encontraremos ningún ejecutable en nuestro *$PATH* que se denomine `exit` o `logout`. Aunque existen diferencias entre ambas órdenes, nosotros no las vamos a contemplar (no son en absoluto triviales), y podremos finalizar una sesión utilizando cualquiera de ellas.

3. ÓRDENES BÁSICAS DE UNIX

3.1. Introducción

No vamos a entrar con mucho detalle en todas las posibilidades y opciones de cada instrucción, debido a que los parámetros de las órdenes pueden variar algo entre clones de Unix (por ejemplo, lo que en Linux puede ser la orden `ps -xua`, en HP-UX el formato es `ps -ef`), y que la potencia de Unix y la gran variedad de opciones de cada orden hacen imposible tratar con detalle cada uno de los posibles parámetros que se pueden dar a un mandato.

En caso de duda sobre una opción determinada, es conveniente consultar el manual *online*, tecleando simplemente *man* <*orden*>; en estas páginas encontraremos todas las características de la orden para el sistema en que nos encontremos.

Aquí se presentan todas las instrucciones que estudiaremos, con mayor o menor profundidad, a lo largo del curso; de cada una de ellas se intenta buscar un equivalente en MS–DOS, ya que sin duda es un entorno (no le llamaremos 'sistema operativo') mucho más familiar para el alumno que Unix. La presentación de estas órdenes Unix sigue una ordenación lógica, no alfabética, dividida en diferentes apartados en función del uso que se da a cada una de ellas. Como veremos, el sistema operativo Unix nos ofrece una gran cantidad de instrucciones orientadas al trabajo con todo tipo de ficheros, en especial con los de texto. Esto no es casual: se suele decir (y no es una exageración) que en un sistema Unix habitual todo son archivos, desde la memoria física del ordenador hasta las configuraciones de las impresoras, pasando por discos duros, terminales, ratones, etc.

3.2. Gestión de archivos y directorios

3.2.1. `ls`

Lista el contenido de directorios del sistema. Existen infinidad de opciones para la orden `ls` (*-a, -l, -d, -r,...*) que a su vez se pueden combinar de muchas formas, y por supuesto varían entre Unices, por lo que lo más adecuado para conocer las opciones básicas es consultar el manual *on-line* (`man ls`). Sin embargo, de todas éstas, las que podríamos considerar más comunes son:

- -l (*long*): Formato de salida largo, con más información que utilizando un listado normal.
- -a (*all*): Se muestran también archivos y directorios ocultos.
- -R (*recursive*): Lista recursivamente los subdirectorios.

Por ejemplo, un resultado típico obtenido al ejecutar `ls` con una opción determinada puede ser el siguiente:

```
rosita:~/programas/prueba$ ls -l
total 5
drwxr-xr-x   2 toni     users        1024 Feb 27 00:06 Cartas
drwxr-xr-x   2 toni     users        1024 Feb 11 16:06 Datos
drwxr-xr-x   2 toni     users        1024 Nov 18 01:36 Programacio
-rwxr-xr-x   2 toni     users        1024 Feb 23 23:58 programa1
-rw-r--r--   1 toni     users         346 Oct 19 18:31 cunix.lista
rosita:~/programas/prueba$
```

La primera columna indica las ternas de permisos de cada fichero o directorio, con el significado que veremos en el capítulo dedicado a conceptos del sistema operativo. El segundo hace referencia al número de enlaces del archivo (aspecto que también comentaremos más adelante), mientras que los dos siguientes indican el propietario y el grupo al que pertenece. El quinto campo corresponde al tamaño del fichero en bytes, y los siguientes dan la fecha y hora de la última modificación del archivo. Obviamente, la última columna indica el nombre del fichero o directorio.

3.2.2. `cd`

La orden `cd` en Unix hace lo mismo que en MS-DOS: nos permite cambiar de directorio de trabajo mediante una sintaxis básica: *cd* <*directorio*>. Las diferencias con el `'cd'` de MS-DOS son mínimas: si en Unix tecleamos `'cd'` sin argumentos iremos a nuestro directorio *$HOME*, mientras que si esto lo hacemos en MS-DOS se imprimirá en pantalla el directorio en que nos encontramos (en Unix esto se consigue con `'pwd'`, como veremos más adelante). Si le pasamos como parámetro una ruta (absoluta o relativa) de un directorio cambiaremos a tal directorio, siempre que los permisos del mismo lo permitan:

```
rosita:~# cd /etc
rosita:/etc# pwd
/etc
rosita:/etc# cd
rosita:~# pwd
/root
rosita:~#
```

La orden `cd` es interna al *shell* en cualquier Unix con que trabajemos: no existirá nunca un ejecutable denominado `'cd'` capaz de cambiar de directorio, ya que por diseño del sistema operativo un proceso no puede modificar ciertas propiedad de sus antecesores.

3.2.3. `pwd`

Imprime en pantalla la ruta completa del directorio de trabajo donde nos encontramos actualmente. No tiene opciones, y es una orden útil para saber en todo momento en qué punto del sistema de archivos de Unix nos encontramos, ya que este sistema es muy diferente a otros operativos como MS-DOS, MacOS, etc. Su ejecución es sencilla:

```
luisa:~$ pwd
/home/toni
luisa:~$ cd /etc/
luisa:/etc$ pwd
/etc
luisa:/etc$
```

3.2.4. `mkdir`

La orden `mkdir` crea (si los permisos del sistema de ficheros lo permiten) el directorio o directorios que recibe como argumentos; aunque esta orden admite diferentes parámetros, no los veremos aquí. Podemos crear directorios de la siguiente forma:

```
rosita:~/tmp$ ls -l
total 0
rosita:~/tmp$ mkdir d1
rosita:~/tmp$ mkdir d2 d3
rosita:~/tmp$ ls -l
total 12
drwxr-xr-x   2 toni     users        4096 May  2 00:43 d1/
drwxr-xr-x   2 toni     users        4096 May  2 00:43 d2/
drwxr-xr-x   2 toni     users        4096 May  2 00:43 d3/
rosita:~/tmp$
```

3.2.5. `touch`

Actualiza la fecha de última modificación de un archivo, o crea un archivo vacío si el fichero pasado como parámetro no existe. Con la opción *-c* no crea este archivo vacío. Su sintaxis es *touch* [*-c*] <*archivo*>:

```
luisa:~/curso$ ls -l
total 4
-rw-r--r--    1 toni    users          5 May 13 18:52 fichero1
luisa:~/curso$ date
Fri May 16 20:53:46 CEST 2003
luisa:~/curso$ touch fichero2 fichero1
luisa:~/curso$ ls -l
total 4
-rw-r--r--    1 toni    users          5 May 16 20:53 fichero1
-rw-r--r--    1 toni    users          0 May 16 20:53 fichero2
luisa:~/curso$
```

3.2.6. `cp`

La orden `‘cp’`, equivalente al `‘copy’` o `‘xcopy’` de MS-DOS, se ejecuta de la forma *cp* <*origen*><*destino*>, y evidentemente copia el archivo indicado en el parámetro ‘origen’ en otro lugar del disco, indicado en ‘destino’; a diferencia de MS-DOS, en Unix es siempre **obligatorio** especificar el destino de la copia. Si este destino es un directorio podemos indicar varios orígenes (tanto archivos como directorios) que serán copiados en su interior; estos múltiples orígenes se pueden definir uno a uno, por su nombre, o bien mediante *wildcards* (comodines), que son los símbolos especiales ‘∗’ (sustituible por uno o más caracteres) y ‘?’ (sustituible por uno solo). Así, la primera de las dos órdenes siguientes copiará los ficheros `‘fichero1’` y `‘fichero2’` en el directorio `‘direc1’`, y la segunda hará lo mismo con todos los archivos que finalicen en `‘.h’`; en ambos casos el directorio destino ha de existir previamente:

```
rosita:~# cp fichero1 fichero2 direc1
rosita:~# cp *.h direc1
rosita:~#
```

Debemos recordar que en Unix el campo ‘.’ no separa el nombre de la extensión de un fichero, como en MS-DOS, sino que es simplemente un carácter más.

Para copiar de forma recursiva podemos utilizar la opción *‘-r’*, que copiará tanto ficheros como directorios completos en un determinado destino; por ejemplo, si queremos copiar todo el directorio `/etc/` en el directorio actual (`‘.’`) lo haremos de la siguiente forma:

```
rosita:~# cp -r /etc .
rosita:~#
```

La orden anterior **no** es igual a esta otra:

```
rosita:~# cp -r /etc/* .
rosita:~#
```

La diferencia entre ambas radica en el origen indicado: en el primer caso le decimos al sistema operativo que copie el directorio y todo lo que cuelga de él, por lo que en el destino se creará un nuevo subdirectorio denominado `‘etc’` y que contendrá todo lo que había en el origen; en cambio, en el segundo caso estamos diciendo que se copie todo lo que cuelga del directorio, pero no el propio directorio, por lo que en el destino se copiarán todos los ficheros y subdirectorios del origen, pero sin crear el subdirectorio `‘etc’`.

3.2.7. `mv`

Renombra un archivo o directorio, o mueve un archivo de un directorio a otro. Dependiendo del uso, su sintaxis variará: *mv* <*archivo/s*> <*directorio*> moverá los archivos especificados a un directorio, y *mv* <*archivo1*><*archivo2*> renombrará el primer fichero, asignándole el nombre indicado en <*archivo2*>. Veamos unos ejemplos:

```
rosita:~# mv hola.c prueba.c
(Renombra el archivo hola.c como prueba.c)
rosita:~# mv *.c direc1
(Mueve todos los archivos finalizados en .c al directorio direc1)
```

3.2.8. `rm`

Elimina archivos o directorios. Sus tres opciones son *-r* (borrado recursivo, de subdirectorios), *-f* (no formula preguntas acerca de los modos de los archivos), y *-i* (interactivo, solicita confirmación antes de borrar cada archivo). Su sintaxis es muy sencilla: *rm* [*-rfi*] <*archivo*>:

```
luisa:~/curso$ ls -la
total 4
drwxr-xr-x    2 toni    users        4096 May 16 20:48 directorio
-rw-r--r--    1 toni    users           0 May 16 20:48 fichero1
-rw-r--r--    1 toni    users           0 May 16 20:48 fichero2
-rw-r--r--    1 toni    users           0 May 16 20:48 fichero3
luisa:~/curso$ rm directorio/
rm: ‘directorio’ is a directory
luisa:~/curso$ rm -r directorio/
luisa:~/curso$ rm -i fichero1
rm: remove ‘fichero1’? y
luisa:~/curso$ rm -f fichero*
luisa:~/curso$ ls -la
luisa:~/curso$ ls -la
total 8
drwxr-xr-x    2 toni    users        4096 May 16 20:49 .
drwxr-xr-x    5 toni    users        4096 May 16 20:47 ..
luisa:~/curso$
```

3.2.9. `rmdir`

Borra directorios **sii** están vacíos, recibiendo como argumento su nombre (o sus nombres, ya que podemos indicar más de un directorio en la misma línea de órdenes).

```
luisa:~/curso$ mkdir d1 d2 d3
luisa:~/curso$ ls -l
total 12
drwxr-xr-x  2 toni toni 4096 Jan 27 20:30 d1
drwxr-xr-x  2 toni toni 4096 Jan 27 20:30 d2
drwxr-xr-x  2 toni toni 4096 Jan 27 20:30 d3
luisa:~/curso$ rmdir d1
luisa:~/curso$ rmdir d2 d3
luisa:~/curso$ ls -l
total 0
luisa:~/curso$
```

Si queremos borrar directorios que no estén vacíos, hemos de utilizar la orden `rm -r`, equivalente a `‘deltree’` en MS–DOS..

3.2.10. `ln`

Asigna (*link*) un nombre adicional a un archivo. Para nosotros, la opción más interesante será *-s*, que nos permitirá crear enlaces simbólicos entre archivos del sistema; analizaremos los dos tipos de enlaces de Unix (enlaces duros y enlaces simbólicos) más adelante. La sintaxis básica de este mandato es *ln* [*-s*] <*fuente*><*destino*>:

```
luisa:~$ ls -l test.c
-rw-r--r--   1 toni     users        1585 Jan  4 00:00 test.c
luisa:~$ ln test.c duro.c
luisa:~$ ln -s test.c simbolico.c
luisa:~$ ls -l test.c duro.c simbolico.c
-rw-r--r--   2 toni     users        1585 Jan  4 00:00 duro.c
lrwxrwxrwx   1 toni     users           6 Feb  3 04:21 simbolico.c -> test.c
-rw-r--r--   2 toni     users        1585 Jan  4 00:00 test.c
luisa:~$
```

3.2.11. `chmod`

Esta orden cambia los permisos de acceso del archivo o directorio que se le pasa como referencia. Existen dos formas básicas de invocarla: la octal y la simbólica. En la primera de ellas se indica en octal el modo deseado para el fichero, de la forma *chmod <modo><archivo>*; se trata de una indicación explícita del permiso deseado para el archivo, ya que estamos marcando **todos** los *bits* de tal permiso (no nos importa el modo anterior del fichero, sea el que sea el nuevo permiso será el indicado en la línea de órdenes):

```
rosita:~$ ls -l prueba
-rw-r--r--   1 toni     users           0 Dec 20 22:54 prueba
rosita:~$ chmod 0 prueba
rosita:~$ ls -l prueba
----------   1 toni     users           0 Dec 20 22:54 prueba
rosita:~$
```

La segunda forma de ejecutar `'chmod'` se denomina simbólica, y en ella se indican únicamente los permisos que cambian con respecto al permiso actual del fichero, de la forma *chmod <who>+|-<permiso><archivo>* (por tanto, el permiso resultante dependerá del que ya tenía el archivo). Indicaremos, en el parámetro *who*, la identidad del usuario/s cuya terna de permisos queremos modificar (*u–user*, *g–group*, *o-others*); a continuación irá un símbolo '+' o un '-', dependiendo si reseteamos el *bit* correspondiente o lo activamos, y en el campo *'permiso'* debemos colocar el permiso a modificar (*r–read*, *w–write*, *x–exec*):

```
rosita:~$ ls -l prueba
-rw-r--r--   1 toni     users           0 Dec 20 22:54 prueba
rosita:~$ chmod g+w prueba
rosita:~$ ls -l prueba
-rw-rw-r--   1 toni     users           0 Dec 20 22:54 prueba
rosita:~$ chmod go-r,o+w prueba
rosita:~$ ls -l prueba
-rw--w--w-   1 toni     users           0 Dec 20 22:58 prueba
rosita:~$
```

Hasta ahora hemos creado ficheros y directorios sin ningún tipo de nociones acerca de sus permisos; no obstante, cuando generamos un archivo éste tiene unos permisos por defecto que nosotros no especificamos. ¿Cuáles son estos permisos? Los permisos con los que creamos ficheros y directorios en un sistema Unix se calculan a partir de una máscara determinada, que podemos obtener o modificar mediante la orden `umask`:

```
anita:~$ umask
0022
anita:~$
```

Este valor (en nuestro caso, 0022) es el número que debemos restar a 0777 para determinar los permisos por defecto de un directorio (0755) y a 0666 en el caso de ficheros planos (0644); podemos modificar nuestra máscara de permisos pasándole a `umask` el nuevo valor:

```
anita:~$ umask 0002
anita:~$ umask
0002
anita:~$
```

En el capítulo reservado a conceptos del sistema operativo Unix se explica con detalle el significado y el formato de los diferentes permisos de un fichero o directorio.

3.3. Ayuda en línea: `man`

La práctica totalidad de los clones de Unix existentes en el mercado incorporan unos manuales *on–line* en los que se describe la sintaxis, opciones y utilidad de las diferentes órdenes del sistema. Este manual es, por supuesto, `man`, que localiza e imprime en pantalla la información solicitada por el usuario.

`man` se ha convertido en una orden indispensable para cualquier usuario, administrador o programador de un sistema Unix. La gran variedad de clones han contribuido fuertemente a esta utilidad del manual, ya que en diferentes sistemas las opciones de un determinado mandato para un mismo fin pueden variar ligeramente (por ejemplo, el `ps -aux` de ciertos sistemas Unix, como Linux, es equivalente al `ps -ef` de otros, como Solaris o HP-UX); incluso en la sintaxis de funciones del lenguaje C, el más utilizado en entornos Unix, puede haber diferencias mínimas entre clones. Conociendo el nombre de la orden o función, y sabiendo manejar mínimamente las páginas del manual, la transición de un Unix a otro no implica ningún problema. La consulta del manual es frecuentemente el camino más corto hacia una solución para una pregunta.

En nuestro sistema Linux, como en otros muchos Unix, las páginas del manual se organizan en diferentes categorías, atendiendo a la clasificación de la llamada sobre la que se intenta buscar ayuda (órdenes de usuario, juegos, llamadas al sistema, funciones de librería, etc.). Cada una de estas categorías se almacena en un directorio diferente del disco; si queremos saber la localización de estos directorios, habremos de visualizar la variable de usuario `$MANPATH`. En algunos Unix, por ejemplo en ciertas versiones de Linux – generalmente antiguas –, tenemos implementado el mandato `manpath`, que nos indicará la ruta donde `man` va a buscar las páginas de ayuda que nosotros le pidamos.

Como hemos comentado varias veces ya, la sintaxis de `man` más simple es *man <orden>*. Sin embargo, tenemos una serie de opciones básicas que es necesario conocer:

`-a`: *All*. Fuerza a `man` a mostrarnos, una a una, todas las diferentes páginas para una misma instrucción o función; por ejemplo, si tecleamos

```
rosita:~# man write
```

el sistema nos mostrará la primera sección de ayuda encontrada (la correspondiente al mandato `write`). Sin embargo, al teclear

```
rosita:~# man -a write
```

el sistema nos va a mostrar todas las páginas de ayuda diferentes que existan (en este caso, la del mandato `write` y la de la función C `write()`).

Si de antemano conocemos la sección que queremos ver, podemos saltar el resto de páginas utilizando *man <seccion><orden>*; si escribimos

```
rosita:~# man 2 write
```

el sistema mostrará directamente la página de ayuda correspondiente a la función `write()`, omitiendo el resto.

`-w`: *Whereis.* No imprime las páginas de ayuda en pantalla, sino que nos indica los archivos, con su ruta absoluta, donde se encuentran tales páginas (que serán ficheros finalizados con la extensión `.gz`, esto es, comprimidos con `gzip` para no ocupar espacio innecesario en el disco). En combinación con '`-a`', nos dará la ruta de todas las páginas que queremos consultar.

`-h`: *Help.* Nos imprime una pantalla de ayuda sobre las opciones básicas de la orden `man`.

Hemos de recordar que `man` es también una orden de Unix, con su propia página de manual. Por lo tanto, la orden `man man` nos dará información sobre la sintaxis completa y todas las opciones de `man`. Por último, saber que las teclas para desplazarnos entre las diferentes pantallas de ayuda son las mismas que para la orden `more`, ya vista.

3.4. Información del sistema

3.4.1. date

Esta orden imprime en pantalla la fecha y la hora en que nos encontramos, algo similar a lo que en MS-DOS se consigue por separado mediante '`time`' y '`date`'. Sus opciones para el usuario normal, que no puede modificar los datos ofrecidos por '`date`', sólo hacen referencia al formato de presentación, por lo que nos limitaremos a invocar esta orden simplemente como `date`, sin ninguna opción:

```
rosita:~# date
Thu Dec 19 04:42:12 CET 1996
rosita:~#
```

3.4.2. uname

Imprime el nombre y algunas características del sistema donde nos encontramos trabajando. Únicamente trataremos la opción *-a* (*all*), que engloba todas las demás; nos indicará el nombre del sistema operativo, el nombre del nodo en la red de comunicaciones, la versión del *kernel* del S.O. y el modelo del microprocesador de la máquina:

```
luisa:~$ uname -a
Linux luisa 2.4.18 #3 Tue Apr 29 00:08:20 CEST 2003 i686 unknown
luisa:~$
```

3.4.3. id

Nos informa sobre nuestro UID (*User IDentifier*, identificador de usuario), y sobre nuestro GID (*Group IDentifier*, identificador de grupo); se trata de dos números que distinguen, en cualquier sistema Unix, a los usuarios y a los grupos (respectivamente) entre sí. El UID ha de ser diferente para cada usuario del sistema, ya que dos usuarios con el mismo identificador serían el mismo para Unix, aunque tuvieran *logins* diferentes; el UID 0 siempre corresponde al administrador). Por su parte, el GID suele ser común para varios usuarios, englobados todos dentro de un mismo grupo que los defina: *users, cursos, programadores. . . .* Aunque generalmente la información que nos va a proporcionar `id` va a consistir simplemente en nuestros UID y GID, en situaciones excepcionales nos va a ofrecer también nuestros EUID y EGID (*Effective UID* y *Effective GID*, respectivamente), si éstos difieren de los anteriores. En este caso puede ser conveniente notificar el hecho al administrador del sistema, ya que suele ser un signo de inestabilidad de la máquina.

La ejecución de esta orden es muy sencilla (no vamos a ver posibles opciones de la instrucción), tal y como se muestra en el siguiente ejemplo:

```
rosita:~$ id
```

```
uid=1000(toni) gid=100(users) groups=100(users)
rosita:~$
```

3.4.4. `who`

Informa sobre quién está conectado en el sistema y ciertas características de dichas conexiones: origen, terminal asignada, hora de conexión, nombre de usuario, etc. de una forma muy similar a '`w`'. No entraremos en sus diferentes opciones, por lo que su sintaxis quedará simplemente `who`:

```
rosita:~$ who
root     tty1     Apr  5 05:12
toni     pts/13   Apr  5 13:01
toni     pts/24   Apr  5 13:16
rosita:~$
```

3.4.5. `w`

Esta orden muestra los usuarios conectados actualmente al sistema, como hemos dicho de una forma parecida a `who`, pero proporcionando más datos que esta última orden:

```
rosita:~$ w
  6:53pm  up 67 days, 10:55,  5 users,  load average: 0.15, 0.11, 0.09
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
toni     tty1     -                28Nov 5  6days 12.23s  0.03s  startx
toni     ttyp0    pc-toni           6:31pm 21:24   0.26s  0.20s  less /tmp/p
toni     pts/4    :0.0             Mon 9am 43:46m 58:04  58:04   ./a.out
toni     ttyp1    pc-toni           6:31pm  0.00s  0.96s  0.83s  vi ordenes.tex
toni     pts/2    :0.0             Thu10pm 19:04m  0.09s  0.09s  -bash
rosita:~$
```

3.4.6. `last`

La orden `last` nos proporciona información acerca de los últimos usuarios que han entrado en el sistema: lugar desde donde conectaron, tiempo de conexión, hora... Podemos especificar opciones (que por supuesto se pueden combinar) para ver los registros correspondientes a un cierto usuario (simplemente indicando su *login*), a un cierto *host* origen (con '`-h host`'), etc., y también limitar el número de registros a mostrar (con '`-N`', siendo '`N`' el número de registros que queremos ver en pantalla); si no se pone ninguna opción la orden imprime un listado de todas las últimas entradas y salidas del sistema (conviene filtrarlo con la orden `more` para poder leer la información con más comodidad):

```
rosita:~$ last -3 toni
toni             pts/4    :0.0             Mon Dec 24 05:55   still logged in
toni             pts/4    :0.0             Mon Dec 24 05:38 - 05:38  (00:00)
toni             pts/3    luna             Mon Dec 24 04:27 - 04:28  (00:00)
rosita:~$
```

3.4.7. `ps`

Informa del estado de los procesos en el sistema (*Process Status*). Sin opciones, `ps` nos informará de los procesos en la sesión de trabajo actual. Con la opción '`-u`' se nos proporciona información más detallada (usuario que posee el proceso, estado del mismo, consumo de memoria y CPU...), mientras que si utilizamos la opción '`-x`' se nos mostrarán todos los procesos de usuario (los de la sesión actual y los de cualquier otra). Si lo que queremos es ver **todos** los procesos lanzados en la máquina, la opción adecuada es '`-aux`' (equivalente a '`-ef`' en algunos sistemas Unix)[2]:

[2] En ciertas versiones de `ps` no es necesario incluir el signo '`-`' antes de las opciones.

```
luisa:~$ ps
  PID TTY          TIME CMD
 4178 pts/6    00:00:00 bash
 4192 pts/6    00:00:00 ps
luisa:~$ ps -u
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
toni      4178  0.0  0.3  1748 1028 pts/6    S    01:00   0:00 -bash
toni      4193  0.0  0.2  2576  944 pts/6    R    01:03   0:00 ps -u
luisa:~$
```

3.4.8. `kill`

La orden `kill` se utiliza para enviarle una señal a un proceso, aunque para nosotros esta orden simplemente matará al proceso; su sintaxis es *kill* [-*sig*] <*PID*>. El PID (*Process IDentifier*, identificador de proceso) de uno de nuestros procesos lo podemos hallar mediante la orden `ps`, y la señal más habitual es la número `9` (SIGKILL), que nos asegura la terminación de un proceso sea cual sea su estado.

Para eliminar a un proceso hemos de obtener en primer lugar su PID mediante `ps` y después utilizar `kill` sobre ese identificador:

```
rosita:~$ ps
  PID TTY          TIME CMD
  889 pts/3    00:00:00 bash
  983 pts/3    00:00:00 sleep
 1416 pts/3    00:00:00 ps
rosita:~$ kill -9 983
rosita:~$ ps
  PID TTY          TIME CMD
  889 pts/3    00:00:00 bash
 1420 pts/3    00:00:00 ps
rosita:~$
```

3.4.9. `du`

La orden `du` (*Disk Usage*) nos dice cuanto ocupa un directorio en y todos sus descendientes (esto es, nos dice lo que ocupan los archivos de ese directorio y todos sus subdirectorios). Su sintaxis es *du* [-*opciones*] [*path*], y las opciones más útiles para nosotros serán las siguientes:

-k: muestra el tamaño en *Kbytes* (aunque en Linux es así por defecto, esta opción es útil en sistemas Unix en los que se muestra el tamaño utilizado en bloques de 512 *bytes*).
-s: muestra total (sin especificar el tamaño de los subdirectorios).

Si no indicamos el parámetro *'path'* se muestra lo que ocupa el directorio actual y todos sus hijos; en el siguiente ejemplo podemos ver que el directorio `~/pruebas` ocupa unos 13 MB:

```
rosita:~/pruebas$ du -sk
13442   .
rosita:~/pruebas$
```

3.4.10. `df`

La orden `df` (*Disk Free*) nos informará acerca del espacio disponible en el sistema de archivos. Para nosotros no existirán parámetros adicionales, sino que simplemente lo ejecutaremos como `df`, y la salida ofrecida será algo así como:

```
rosita:~# df
Filesystem         1024-blocks    Used Available Capacity  Mounted on
/dev/hdb1               297842  206584     75362      73%  /
/dev/hda1               614672  146192    468480      24%  /mnt
rosita:~#
```

Vemos que el sistema tiene dos discos ubicados en lugares diferentes. El primero tiene aproximadamente 300 MB, de los cuales mas de 200 están ocupados (el 73 % del disco). El segundo, de unos 615 MB, tiene disponibles alrededor de 470 MB.

3.5. Tratamiento básico de archivos

3.5.1. `file`

La orden `file` nos proporciona información sobre el tipo del archivo (o archivos) especificados como argumento. Es necesario recordar que en Unix no existe el concepto de extensión, por lo que el hecho de que el nombre de fichero acabe en '`.txt`' o en '`.c`' no dice nada a priori acerca del mismo: podemos tener un archivo ejecutable que se llame '`carta.txt`', un directorio que se llame '`prueba.doc`' o un fichero de texto denominado '`...`'. Por supuesto, nadie suele poner estos nombres – no relacionados con su contenido – a sus ficheros o directorios, ya que si su número es alto la gestión de los mismos sería difícil, pero el sistema operativo no pone ninguna restricción si queremos hacerlo.

A continuación se muestran algunos ejemplos de la ejecución de '`file`'; podemos ver que nos informa que el archivo '`hola.c`' es en realidad un directorio, que '`prueba.txt`' es un binario ejecutable, y que '`vacio`' es un fichero vacío. Además, no puede determinar el formato del último archivo, por lo que simplemente nos dice que se trata de datos ('`data`'):

```
rosita:~$ file hola.c
hola.c: directory
rosita:~$ file prueba.txt
prueba.txt: ELF 32-bit LSB executable, Intel 80386, version 1,\
dynamically linked (uses shared libs), stripped
rosita:~$ file vacio
vacio: empty
rosita:~$ file nolose
nolose: data
rosita:~$
```

3.5.2. `cat`

La orden '`cat`' es equivalente al '`type`' de MS-DOS; es decir, su función es mostrar en pantalla el contenido de un archivo:

```
rosita:~# cat hola.c
#include <stdio.h>
main(){
    printf("Hola, mundo\n");
}
rosita:~#
```

También podemos utilizar `cat` para unir un grupo de archivos de texto en uno solo, tanto por pantalla como – más habitual – redireccionando la salida a un fichero, tal y como veremos más adelante:

```
rosita:~# cat fichero1
Este es el primer fichero
```

```
rosita:~# cat fichero2
Este es el segundo fichero
rosita:~# cat fichero1 fichero2
Este es el primer fichero
Este es el segundo fichero
rosita:~#
```

3.5.3. `more/less`

Visualiza un archivo pantalla a pantalla, no de forma continua. Es una especie de *cat* con pausas, que permite una cómoda lectura de un archivo. Al final de cada pantalla nos aparecerá un mensaje indicando `--More--`. Si en ese momento pulsamos **Intro**, veremos una o más líneas del archivo; si pulsamos la barra espaciadora, veremos la siguiente pantalla, si pulsamos **b** la anterior, y si pulsamos **q** saldremos de `more`. Su sintaxis es *more* <*archivo*>.
La orden `less` es muy similar a `more` pero con una ventaja sobre esta: permite recorrer el texto tanto hacia delante como hacia atrás tantas veces como queramos.

3.6. Tratamiento avanzado de archivos

3.6.1. `head`

La orden `head` muestra las primeras líneas (10 por defecto) de un archivo que recibe como parámetro. La principal opción de este mandato es la que especifica el número de líneas a visualizar: (`'-n'`), y su sintaxis es *head* [*-n numero*] <*archivo*>, o simplemente *head* [*-numero*] <*archivo*>:

```
rosita:~$ head -3 ejemplo.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
rosita:~$
```

3.6.2. `tail`

Visualiza las últimas líneas de un archivo (es la instrucción 'contraria' a `head`). Podemos especificar (como en `head`) el número concreto de líneas a mostrar mediante el parámetro *-n* <*lineas*> o simplemente -<*lineas*>; si no indicamos este valor de forma explícita, `tail` mostrará por defecto las diez últimas líneas del archivo.

La sintaxis de esta orden es *tail* [*-opcion*] *archivo*:

```
luisa:~$ cat prueba
En un lugar de la Mancha
de cuyo nombre no quiero acordarme...
luisa:~$ tail -1 prueba
de cuyo nombre no quiero acordarme...
luisa:~$
```

3.6.3. `cmp`

Esta orden compara el contenido de dos archivos, imprimiendo en pantalla la primera diferencia encontrada (si existe); recibe como parámetros los dos nombres de los ficheros a comparar:

```
rosita:~# cmp prova.c prova2.c
rosita:~# cmp prova.c prova3.c
prova.c prova3.c differ: char 10, line 3
rosita:~#
```

En el primer caso, si no se produce ninguna salida, ambos ficheros son idénticos; en el segundo se muestra que la primera diferencia está en el décimo carácter de la tercera línea.

3.6.4. `diff`

Esta orden compara dos archivos, indicándonos las líneas que difieren en uno con respecto al otro (si ambos son iguales, la ejecución no producirá ninguna salida en pantalla). Su sintaxis es *diff <fichero1><fichero2>*:

```
rosita:~# diff file1 file2
rosita:~#
rosita:~# diff file1 file3
2c2
< de cuyo nombre ...
---
> de cuyo nobre ...
rosita:~#
```

En el primer caso tenemos dos ficheros exactamente iguales, y en el segundo dos que son también muy parecidos, pero que uno de ellos presenta un pequeño error que los hace diferentes.

¿Por qué hablamos de `diff` si ya conocíamos la orden `cmp`, que también nos sirve para comparar archivos y ver si son iguales o no? Aunque a primera vista no nos lo parezca, ambas instrucciones son muy diferentes: sin ir más lejos, `diff` recorre ambos ficheros por completo, mientras que `cmp` se detiene nada más encontrar una diferencia (si quisiéramos simplemente comprobar que dos archivos de 5 GB son diferentes, ¿qué orden usaríamos de las dos vistas?). Además, la programación de `cmp` es muy simple, mientras que `diff` incorpora algoritmos complejos de comparación de cadenas y subcadenas.

`diff` se suele utilizar para distribuir nuevas versiones de algún código fuente, por ejemplo el del núcleo de Linux: si tenemos la versión 1 de un código y necesitamos la 2, o bien la descargamos completa o bien conseguimos únicamente las líneas que difieren entre ambas, de forma que podemos modificar la versión antigua para transformarla en la nueva (por ejemplo con una orden como `patch`, que no vamos a ver aquí).

3.6.5. `grep`

Esta es quizás una de las órdenes más utilizadas e importantes en cualquier sistema Unix; `grep` busca un patrón (que recibe como parámetro) en un archivo. Su sintaxis es sencilla: *grep [opt] <patron> <fichero>*; al tratarse de una potente herramienta de Unix muchas de sus opciones son de interés para nosotros, en particular '`-v`' (imprime las líneas que no coinciden con el patrón), '`-i`' (no distingue mayúsculas/minúsculas) y '`-c`' (imprime la cuenta de coincidencias).

`grep` se suele utilizar tanto como instrucción directa para buscar una cadena en un fichero como unido a otras órdenes para filtar su salida, mediante '|'); por ejemplo, las dos órdenes siguientes son equivalentes:

```
rosita:~# grep -i include prueba.c
#include <stdio.h>
#include <stdlib.h>
rosita:~# cat prueba.c|grep -i include
#include <stdio.h>
#include <stdlib.h>
rosita:~#
```

3.6.6. `wc`

Es un contador de líneas, palabras y caracteres (*Word Count*). Es posible utilizarlo como mandato directo y como filtro, y también es una potente herramienta en *shellscripts*. Su sintaxis es *wc* [*-opcion*] <*archivo*>, siendo *opcion* una de las siguientes:

- `-l`: cuenta líneas.
- `-c`: cuenta caracteres
- `-w`: cuenta palabras.

Es igualmente posible utilizar combinaciones de las tres opciones; por defecto, `wc` asumirá todas: `-lwc`. Si no se indica un nombre de archivo, la orden espera datos de la entrada estándar (el teclado):

```
luisa:~$ wc /etc/passwd
     23      34     908 /etc/passwd
luisa:~$ wc -l /etc/passwd
     23 /etc/passwd
luisa:~$
```

3.6.7. `sort`

Ordena, compara o mezcla líneas de archivos de texto que recibe como parámetro, clasificándolas según un patrón especificado; a nosotros únicamente nos va a interesar este mandato para ordenar alfabéticamente las líneas de un fichero de texto:

```
rosita:~$ cat telefo.1
Luis    977895
Antonio 3423243
Juan    3242143
rosita:~$ cat telefo.2
Amalio  332210
Joaquin 234234
Pepa    336544
Ana     91-555234
rosita:~$ sort telefo.1
Antonio 3423243
Juan    3242143
Luis    977895
rosita:~$ sort telefo.1 telefo.2
Amalio  332210
Ana     91-555234
Antonio 3423243
Joaquin 234234
Juan    3242143
Luis    977895
Pepa    336544
rosita:~$
```

Una opción de esta orden que nos puede resultar útil es `-f`, que provoca que `sort` interprete las letras minúsculas como mayúsculas.

3.6.8. `spell`

Compara las palabras del archivo especificado con las de un diccionario en Inglés (por defecto, aunque hay diccionarios en castellano y también para textos creados con LaTeX) y nos indica de los

posibles errores gramaticales. A este mandato no suele ser útil indicarle un archivo en castellano (obviamente si estamos trabajando con un diccionario en otra lengua), o con muchos nombre propios o tecnicismos, ya que nos dará contínuamente una serie de errores que serán inexistentes.

La sintaxis que utilizaremos para la orden `spell` es muy sencilla (no habremos de especificar ni tan siquiera el diccionario a usar): *spell <archivo>*. Tras la ejecución, cualquier palabra que no se encuentre en la base de datos del sistema será susceptible de ser sustituida o aceptada por el usuario. La orden `spell` es un primitivo corrector ortográfico, pero de tal potencia y facilidad de uso que aún se encuentra en la mayoría de sistemas Unix del mundo.

3.7. Órdenes de búsqueda

3.7.1. find

La orden `find` localiza archivos en el sistema de ficheros de la máquina que cumplan un determinado patrón que la orden recibe como parámetro; su sintaxis básica es *find <ruta> <patron>*. `find` realiza una búsqueda recursiva a partir del directorio indicado en el campo *'ruta'*, por lo que si queremos que realmente intente localizar un fichero en todo el sistema de archivos habremos de indicarle como parámetro el directorio raíz (*root*, '/'), y tener en cuenta que no podemos buscar en directorios en los que no tenemos los permisos adecuados.

Dentro del campo *'patron'* podemos indicarle a la orden que realice una búsqueda en base al nombre del fichero, a sus permisos, a su tipo... Algunos parámetros interesantes de búsqueda pueden ser los siguientes:

- `-name nombre`: Busca ficheros o directorios cuyo nombre coincida con el especificado como parámetro; podemos utilizar comodines ('?' y ∗) si no conocemos el nombre exacto del fichero a buscar. Por ejemplo, para buscar todos los ficheros (y directorios) cuyo nombre finalice en '.tex' a partir del directorio actual podemos utilizar la orden siguiente:

```
rosita:~$ find . -name "*.tex"
./prueba.tex
./documentos/cunix.tex
./Unix/admin.tex
rosita:~$
```

- `-empty`: Busca ficheros o directorios vacios; esto nos puede resultar útil para eliminar ficheros temporales sin contenido que alguna aplicación haya podido dejar por el sistema de ficheros:

```
rosita:~$ find . -emtpy
./directorio1
./vacio
./prueba/core
rosita:~$ rmdir directorio1
rosita:~$ rm vacio prueba/core
rosita:~$
```

- `-type tipo`: Busca ficheros en función de su tipo, que puede ser 'f' (ficheros planos), 'd' (directorios), 'l' (enlaces simbólicos), etc. Por ejemplo, para buscar todos los directorios que 'cuelgan' de *$HOME* podemos utilizar la siguiente orden:

```
rosita:~$ find $HOME -type d
/home/toni/Unix
/home/toni/documentos
/home/toni/Seguridad
/home/toni/temporal
rosita:~$
```

3.7.2. `which`

La instrucción `which` buscará el archivo o archivos especificados como argumento en todos los directorios incluidos en nuestra variable de entorno *$PATH*, diciéndonos dónde es posible localizarlo, o indicando que el fichero no se encuentra en ningún directorio del *$PATH*. Por ejemplo, si queremos saber qué fichero se ejecuta realmente cuando en el *prompt* tecleamos '`ls`', teclearíamos una orden como la siguiente:

```
rosita:~$ which ls
/bin/ls
rosita:~$
```

3.8. Compresión y empaquetado

3.8.1. `gzip`

La orden `gzip` es un compresor de archivos; recibe como parámetro el archivo o archivos a comprimir, y cada uno de ellos es guardado con un '`.gz`' añadido a su nombre original. De la multitud de opciones existentes, que crecen en cada versión del programa, nos interesan principalmente: '`-d`' (para la orden `gzip`, descomprime análogamente a `gunzip`), '`-h`' (muestra la pantalla de ayuda), '`-r`' (movimiento de la estructura de directorios de forma recursiva), '`-t`' (testea la integridad de los archivos comprimidos) y '`-v`' (muestra el nombre y porcentaje de cada archivo comprimido o descomprimido). Si por ejemplo queremos comprimir el fichero `prueba.txt` lo haremos así:

```
rosita:~$ ls -l prueba.txt
-rw-r-----   1 toni     users       11112 Dec 20 02:19 prueba.txt
rosita:~$ gzip prueba.txt
rosita:~$ ls -l prueba.txt.gz
-rw-r-----   1 toni     users       5122 Dec 20 02:19 prueba.txt.gz
rosita:~$
```

Para descomprimirlo después ejecutaríamos la orden de la siguiente forma:

```
rosita:~$ ls -l prueba.txt.gz
-rw-r-----   1 toni     users       5122 Dec 20 02:19 prueba.txt.gz
rosita:~$ gzip -d prueba.txt.gz
rosita:~$ ls -l prueba.txt
-rw-r-----   1 toni     users       11112 Dec 20 02:19 prueba.txt
rosita:~$
```

Es necesario recordar que `gzip` sólo comprime archivos individuales, no crea paquetes comprimidos con varios ficheros en su interior; para conseguir esto debemos utilizar otros programas de compresión, como `zip`, o bien – más habitual – utilizar el empaquetador `tar` junto a `gzip`.

3.8.2. `tar`

La orden `tar` es un empaquetador que enlaza ficheros y subdirectorios en un único archivo (**no** comprime); sus opciones más habituales son '`c`' (crea un contenedor, un paquete), '`x`' (extrae de un contenedor), '`v`' (modo *verbose*), '`t`' (comprueba la integridad de un contenedor) y '`f`' (especifica el nombre del contenedor del cual extraer ficheros o en el que se almacenarán los ficheros).

Vamos a ver cómo crear paquetes y como extraer la información de los mismos mediante `tar`, sin entrar en el resto de opciones -que son muchas- de la orden. Para crear un contenedor, utilizaremos la herramienta de esta forma:

```
[toni@bruja ~/tmp]$ ls -l
total 8
-rw-r--r--  1 toni  toni  1776 24 feb 20:14 fichero1
```

```
-rw-r--r--  1 toni  toni  1511 24 feb 20:14 fichero2
-rw-r--r--  1 toni  toni  3478 24 feb 20:14 fichero3
[toni@bruja ~/tmp]$ tar cvf paquete.tar *
a fichero1
a fichero2
a fichero3
[toni@bruja ~/tmp]$ ls -l
total 18
-rw-r--r--  1 toni  toni  1776 24 feb 20:14 fichero1
-rw-r--r--  1 toni  toni  1511 24 feb 20:14 fichero2
-rw-r--r--  1 toni  toni  3478 24 feb 20:14 fichero3
-rw-r--r--  1 toni  toni  9728 24 feb 20:15 paquete.tar
[toni@bruja ~/tmp]$
```

Como vemos, especificamos a `tar` las opciones '`c`' (crear), '`v`' (modo *verbose*) y '`f`' (nombre del paquete); es importante que este último argumento vaya al final, para ir seguido del nombre del paquete a crear. Como último argumento, indicamos los ficheros a incluir en el nuevo paquete.

Para desempaquetar, deberemos utilizar las mismas opciones pero indicando una '`x`' en lugar de una '`c`':

```
[toni@bruja ~/tmp]$ ls -l
total 10
-rw-r--r--  1 toni  toni  9728 24 feb 20:15 paquete.tar
[toni@bruja ~/tmp]$ tar xvf paquete.tar
x fichero1
x fichero2
x fichero3
[toni@bruja ~/tmp]$ ls -l
total 18
-rw-r--r--  1 toni  toni  1776 24 feb 20:14 fichero1
-rw-r--r--  1 toni  toni  1511 24 feb 20:14 fichero2
-rw-r--r--  1 toni  toni  3478 24 feb 20:14 fichero3
-rw-r--r--  1 toni  toni  9728 24 feb 20:15 paquete.tar
[toni@bruja ~/tmp]$
```

Es habitual encontrar paquetes `tar` comprimidos además con `gzip` (insistimos de nuevo: `tar` no comprime los datos); la versión GNU de la herramienta `tar` incorpora la opción '`z`' que nos permite trabajar con paquetes comprimidos mediante `gzip`. Así, podemos crear paquetes comprimidos o leer de paquetes comprimidos añadiendo una '`z`' a las opciones que hemos visto antes:

```
[toni@bruja ~/tmp]$ ls -l
total 8
-rw-r--r--  1 toni  toni  1776 24 feb 20:14 fichero1
-rw-r--r--  1 toni  toni  1511 24 feb 20:14 fichero2
-rw-r--r--  1 toni  toni  3478 24 feb 20:14 fichero3
[toni@bruja ~/tmp]$ tar cvzf paquete.tar.gz *
a fichero1
a fichero2
a fichero3
[toni@bruja ~/tmp]$ ls -l
total 10
-rw-r--r--  1 toni  toni  1776 24 feb 20:14 fichero1
-rw-r--r--  1 toni  toni  1511 24 feb 20:14 fichero2
-rw-r--r--  1 toni  toni  3478 24 feb 20:14 fichero3
-rw-r--r--  1 toni  toni   967 24 feb 20:22 paquete.tar.gz
```

```
[toni@bruja ~/tmp]$
```

Como vemos, a diferencia de un contenedor `tar`, en este caso el fichero resultante **sí** aparece comprimido, siendo su tamaño menor que el anterior.

3.8.3. bzip2

Al igual que `gzip`, la orden `bzip2` es un compresor de archivos que recibe como parámetro el archivo o archivos a comprimir, y cada uno de ellos es guardado con un '`.bz2`' añadido a su nombre original. De todas sus opciones, muy similares a las de `gzip` nos interesan principalmente '`-d`' (descomprime análogamente a `bunzip2`), '`-h`' (muestra la pantalla de ayuda) y '`-t`' (testea la integridad de los archivos comprimidos). Si por ejemplo queremos comprimir y descomprimir el fichero `piensa.ps`, lo haremos así:

```
[toni@bruja ~]$ ls -l piensa.ps
-rw-r--r--  1 toni  toni  428814  9 sep 19:59 piensa.ps
[toni@bruja ~]$ bzip2 piensa.ps
[toni@bruja ~]$ ls -l piensa.ps.bz2
-rw-r--r--  1 toni  toni  38850  9 sep 19:59 piensa.ps.bz2
[toni@bruja ~]$ bzip2 -d piensa.ps.bz2
[toni@bruja ~]$ ls -l piensa.ps
-rw-r--r--  1 toni  toni  428814  9 sep 19:59 piensa.ps
[toni@bruja ~]$
```

3.9. Otras órdenes

3.9.1. echo

Esta orden imprime sus argumentos en la salida estándar. Aunque `echo` reconoce algunos caracteres especiales (\f,\t,...), no los utilizaremos, limitando su uso al más común: por una parte, imprimir el mensaje especificado como argumento (esta opción se usa generalmente en *shellscripts*, ya que en otros casos no tiene mucha utilidad visualizar el mensaje en pantalla), y por otra consultar el valor de variables de entorno:

```
rosita:~# echo "Hola otra vez"
Hola otra vez
rosita:~# echo $HOME
/home/toni
rosita:~#
```

3.9.2. cal

Esta orden nos mostrará en pantalla el calendario del mes (1–12) y año (1-9999) que recibe como parámetros; si se invoca sin argumentos, muestra el calendario del mes actual. Por defecto, `cal` muestra como primer día de la semana el domingo (`Su`); si queremos que este día sea el lunes, especificaremos la opción '`-m`'.

```
rosita:~$ cal 2 1998
    February 1998
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28

rosita:~$
```

3.9.3. `passwd`

Como vimos en el Capítulo 2, la orden `passwd` se utiliza para cambiar la clave de acceso al sistema. Cuando cambiemos esta contraseña, habremos de repetir nuestra nueva clave dos veces (no se mostrará en pantalla), y en ambas ha de coincidir; esto se realiza para evitar que un error al mecanografiar nos asigne un *password* que no queríamos.

3.9.4. `clear`

La orden '`clear`' limpia el texto de la pantalla, de una forma similar a la instrucción '`cls`' en MS-DOS.

3.9.5. `sleep`

Suspende la ejecución durante un tiempo especificado como argumento, en segundos: *sleep* <*tiempo*>. Es útil en *shellscripts*, no como orden interactiva, para ejecutar una determinada orden después de un cierto intervalo de tiempo.

3.9.6. `nohup`

Mantiene la ejecución de órdenes aunque se desconecte del sistema, ignorando señales de salida y/o pérdida de terminal. Su sintaxis es *nohup* <*orden*>. Nos permite dejar trabajos realizándose aunque no estemos físicamente conectados al ordenador, como sesiones *FTP* largas, compilación de grandes programas por módulos, etc.

3.9.7. `alias`

La orden `alias` nos permite definir una especie de pseudónimo (un *alias*) para alguna instrucción (opciones del mandato incluidas); si tecleamos esta orden sin ningún otro argumento se nos ofrecerá una lista de los pseudónimos que ya tenemos definidos:

```
rosita:~$ alias
alias d='dir'
alias dir='/bin/ls $LS_OPTIONS --format=vertical'
alias ls='/bin/ls $LS_OPTIONS'
alias v='vdir'
alias vdir='/bin/ls $LS_OPTIONS --format=long'
rosita:~$ prueba
bash: prueba: command not found
rosita:~$ alias prueba="echo HOLA"
rosita:~$ prueba
HOLA
rosita:~$
```

Generalmente este mandato se usa para no tener que teclear órdenes largas que se pueden utilizar bastante a lo largo de una sesión. Por ejemplo, imaginemos que solemos conectar desde nuestro sistema Unix a otra máquina (como *ejemplo.upv.es*) por terminal remota. La orden a utilizar sería la siguiente:

```
rosita:~# telnet ejemplo.upv.es
```

Si hemos de realizar esta tarea demasiadas veces, puede ser útil definir un *alias* del siguiente modo:

```
rosita:~# alias telej="telnet ejemplo.upv.es"
```

Así, cada vez que tecleemos `telej`, el sistema interpretará la orden a la que hemos asignado este *alias*.

Los *aliases* son una característica del intérprete de órdenes (`bash` en nuestro caso), y desaparecen al morir éste, por ejemplo finalizando una sesión. Así, puede resultarnos útil incluir unas líneas con nuestros *aliases* en el archivo `.profile`, a fin de tenerlos definidos en cada una de las sesiones en que vayamos a trabajar sin necesidad de volver a teclearlos. Otra forma de eliminar un *alias*, sin tener que salir y volver a entrar en el sistema, es mediante la orden `unalias`:

```
rosita:~# alias prueba
alias prueba='echo HOLA'
rosita:~# unalias prueba
rosita:~# alias prueba
bash: alias: `prueba' not found
rosita:~#
```

4. COMUNICACIÓN ENTRE USUARIOS

4.1. Introducción

En un mismo sistema Unix generalmente estarán conectados varios usuarios a la vez, tal vez físicamente muy lejos, pero lógicamente compartiendo los recursos del ordenador. Existen ocasiones en que nos interesa una comunicación rápida y en tiempo real (escribiendo y leyendo ambas partes al mismo tiempo) con un usuario por cualquier motivo, ya sea para enviar un mensaje urgente, para pedir una información, o simplemente para charlar un poco.

Para este fin, los dos mandatos básicos que suele proporcionar cualquier Unix son `write` y `talk`. Una tercera orden que veremos más adelante, `mail`, para enviar y gestionar primitivamente correo electrónico, no trabaja en tiempo real, pero su sencillez hace interesante el conocer su uso para comunicarnos con otros usuarios de nuestro sistema o de cualquier otro conectado a una red.

4.2. La orden `write`

La orden *write* escribe a otro usuario de nuestro mismo sistema. Copia las líneas desde nuestro terminal al terminal del usuario especificado en la sintaxis del mandato: *write* <*usuario*><*terminal*>. El campo <*terminal*> nos será útil sólo si el usuario con el que queremos comunicarnos tiene varias sesiones abiertas en la máquina.

Imaginemos que queremos comunicarnos con el usuario *luis*. Una vez nos hayamos cerciorado que está conectado (con las órdenes `w` o `who`), teclearemos

```
rosita:~# write luis
```

En la pantalla del terminal de *luis*, aparecerá el mensaje

```
Message from root at tty1
```

Si *luis* quiere comunicarse con nosotros, tecleará a su vez

```
rosita:~$ write root tty1
```

y ya tendremos establecida la conversación. Podemos empezar a teclear nuestros mensajes, hasta que uno de los dos interlocutores pulse *Ctrl-D*, para terminar la conversación o se envíe una interrupción (*Ctrl-C*).

Hemos de recordar que si no deseamos recibir mensajes de otros usuarios, debemos utilizar la orden `mesg n`.

4.3. La orden `talk`

A diferencia de `write`, `talk` nos va a permitir por norma general entablar una conversación con un usuario de otra máquina, no sólo de nuestro propio sistema. Este programa divide la pantalla en dos mitades; en la superior iremos escribiendo nosotros, y en la inferior podremos leer los mensajes de nuestro interlocutor (lo que nosotros escribamos, a él le aparecerá en la pantalla inferior, obviamente).

Imaginemos que estamos en nuestra máquina, *servidor.upv.es*, y queremos comunicarnos con el usuario `root`, del sistema *rosita* (si en lugar de con un usuario en otro ordenador quisiéramos hablar con alguien de nuestro propia máquina, simplemente eliminaríamos el nombre del *host* de las instrucciones).
Para entablar esta comunicación, teclearemos

```
servidor:~$ talk root@rosita
```

Si el usuario `root` de *rosita* está conectado en esos momentos, le aparecerá un mensaje parecido a

```
Message from Talk_Daemon@servidor at 03:57
talk: connected requested by toni@servidor
talk: respond with: talk toni@servidor
```

Si el usuario `root` decide contestarnos, tecleará

```
rosita:~# talk toni@servidor
```

y ya tendremos establecida la conexión.

Existe un mandato similar a éste, pero algo más sofisticado. Se llama *ytalk*, y su utilización es análoga a la de `talk`.

4.4. La orden `mail`

Esta instrucción se usa para el envío y recepción de correo entre usuarios. Aunque hoy en día, bajo sistemas Unix, su uso se ha reducido notablemente debido a otros gestores de correo más amigables, como `elm` y `pine` (que veremos más adelante), no está de más conocer algunas opciones básicas para poder gestionar nuestro correo con `mail`, de una forma rápida y eficiente.

Veremos primero la llamada a `mail` especificando una dirección *e-mail* como argumento (p.e., `mail root@servidor`). Este modo se utilizará para enviar una carta a la dirección especificada. En pantalla nos aparecerá la opción *Subject:* (que podíamos haber especificado en la línea de órdenes con una instrucción como `mail -s <subject> root@servidor`). El *subject* de un *e-mail* es el título, el tema tratado, algo que resuma en una línea el contenido del mensaje...

Después de introducir el *subject* (entre comillas si es compuesto), podemos comenzar a escribir nuestra carta sin más. Cuando hayamos finalizado con ella, teclearemos un punto (.) en la última línea para indicar el final de la transmisión (*EOT, End of Transmission*), y el sistema se encargará de enviar nuestro mensaje al usuario especificado. Veamos un ejemplo sencillo:

```
rosita:~# mail root@rosita
Subject: Test de correo
Hola !!
Esto es solo una prueba del uso de mail.
.
EOT
rosita:~#
```

La segunda forma de utilizar `mail` que vamos a ver es la que trata básicamente la gestión (ordenación, lectura, borrado...) del correo que nos hayan enviado a nosotros, y que estará almacenado en nuestro buzón. Para ello, ejecutaremos `mail` sin argumentos. A partir de aquí, podremos empezar a introducir órdenes `mail` al programa. Éstas órdenes tienen el formato

<orden><lista de mensajes><argumento/s>

La lista de mensajes es el conjunto de mensajes a los que vamos a aplicar la orden dada, y la podemos especificar como

- *n*: Mensaje número *n*.
- .: Mensaje en curso (vendr marcado por > en el entorno de *mail*).
- *$*: Último mensaje.
- ∗: Todos los mensajes.

- *:n*: Todos los mensajes nuevos.
- *:u*: Todos los mensajes no leídos.

Las órdenes `mail` que utilizaremos son las siguientes:

- *?*: Visualizar el resumen de órdenes.
- *copy* <*lista*><*archivo*>: Copia los mensajes de <*lista*> en el archivo especificado.
- *delete* <*lista*>: Borra los mensajes de <*lista*> del buzón.
- *edit* <*lista*>: Edita los mensajes especificados.
- *exit*: Sale de *mail* sin efectuar cambios.
- *file* <*archivo*>: Lee en el archivo especificado.
- *quit*: Sale de *mail*.
- *undelete* <*lista*>: Recupera los mensajes indicados.
- *write* <*lista*><*archivo*>: Graba los mensajes de <*lista*> en el archivo especificado.

5. EL EDITOR DE TEXTOS `vi`

5.1. Introducción

Cualquier fichero de texto (un documento, el código fuente de un programa. . .) puede ser creado, visualizado y modificado utilizando un editor de textos. En la mayoría de sistemas Unix existe una gran variedad de editores disponibles para el usuario: `pico, emacs, joe, ed, vi...` Nosotros nos vamos a centrar en el último, considerado el más estándar dentro de Unix, y por extensión también en `ed`. `vi` estará disponible en cualquier clon del sistema operativo, desde Minix corriendo en un 8086 hasta Unicos sobre una Cray; el resto no son tan usuales, por lo que no los trataremos aquí.

`vi` es un editor de textos orientado a pantalla completa, mientras que `ed` es un editor de líneas (lo que sería el antiguo *edlin* en los sistemas MS-DOS). Con él vamos a poder visualizar el contenido del archivo de texto en pantalla, y el cursor nos va a indicar nuestra posición dentro del fichero para modificarlo. `vi` es un editor de textos demasiado potente para conocer su uso en unas horas; aquí vamos a intentar aprender a realizar las operaciones básicas sobre archivos, pero hemos de insistir en que la mejor forma de saber utilizarlo y aprovechar al máximo sus prestaciones es practicando su uso mucho tiempo. Se suele decir que manejar `vi` es como tocar el piano: por muchos libros o documentos que leamos, no aprenderemos a manejar el editor hasta que no lo utilicemos.

`vi` tiene dos modos de operación diferentes: el modo comando, en el que las pulsaciones de teclas se interpretan no como caracteres sino como órdenes al editor, y el modo edición o inserción, en el que podremos introducir el texto deseado, dependiendo de la orden que se esté ejecutando. Para cambiar entre modos utilizaremos la tecla ESC; sin importar que se esté haciendo, `vi` se pone en modo comando y queda listo para recibir órdenes. De esta forma, cuando no recordemos el modo en el que estamos, bastará con pulsar `ESC` (mucha gente pulsa `ESC--ESC` para cerciorarse que llega al modo comando) y poder seguir trabajando.

5.2. Comenzando con `vi`

Para comenzar a trabajar con un fichero lo habitual es teclear `vi` <`fichero`>; de esta forma, el contenido se volcará en pantalla; obviamente, si es un nuevo archivo, éste estará vacío. De cualquier modo, el cursor se situará en la primera línea, indicando que es la línea actual.

Hemos de resaltar que al editar un archivo simplemente visualizamos una copia en memoria de ese archivo; no se va a modificar hasta que grabemos los cambios con la opción adecuada, que veremos más adelante.

Nada más entrar en `vi`, estaremos en el modo comando del editor. Podremos desplazarnos por el contenido del fichero utilizando las teclas de movimiento del cursor (llamadas también cursores, indicadas con flechas), o alguna de las siguientes opciones:

- Movimiento entre caracteres:
  h: Desplaza el cursor un carácter a la izquierda.
  l: Desplaza el cursor un carácter a la derecha.
- Movimiento entre líneas:
  k: Desplaza el cursor una línea hacia arriba.
  j: Desplaza el cursor una línea hacia abajo.
  H: Desplaza el cursor a la primera línea que se ve en pantalla.
  L: Desplaza el cursor a la última línea que se ve en pantalla.
  M: Desplaza el cursor a la línea que vemos en la mitad de la pantalla.
  <linea>G: (goto) Desplaza el cursor a la línea indicada por el número de *:<linea>*. En versiones de `vi`, basta teclear el número a continuación de los dos puntos (:) para ir directamente a esa línea.

- Movimiento dentro de una línea:
  w: Desplaza el cursor una palabra hacia delante.
  e: Desplaza el cursor al final de la palabra.
  b: Desplaza el cursor al principio de la palabra.
  0: Mueve el cursor al principio de la línea actual.
  $: Mueve el cursor al final de la línea actual.

- Para ver otro trozo diferente del fichero editado, podemos utilizar las órdenes siguientes:
  Ctrl-u: Desplaza la ventana media pantalla hacia abajo.
  Ctrl-d: Desplaza la ventana media pantalla hacia arriba.
  Ctrl-f: Desplaza la ventana una pantalla hacia abajo.
  Ctrl-b: Desplaza la ventana una pantalla hacia arriba.
  G: Sitúa el cursor en la última línea del documento.

5.3. Saliendo del editor

Las modificaciones que hemos realizado con `vi` en un fichero de texto no se han realizado sobre el mismo fichero, sino sobre una copia que el sistema ha almacenado en la memoria principal. Al salir del editor, podemos elegir grabar o no los cambios en el archivo, con uno de los siguientes mandatos:

- **:x**: Graba y sale al intérprete de órdenes. Hemos de pulsar *Intro* para ejecutar el mandato.
- **:wq**: Al igual que `:x`, graba los cambios y sale del editor. Tambin hemos de pulsar *Intro* al final.
- **ZZ**: Graba y sale, pero sin necesidad de pulsar *Intro*.
- **:q!**: Sale del editor y descarta todos los cambios realizados sobre el archivo (no se grabará ninguna modificación perceptible por el usuario).

5.4. Tratamiento del texto

Para añadir texto nuevo en un archivo primero hemos de situar el cursor en la posición deseada (recordad que por defecto *vi* lo situará en la primera línea y la primera columna), con las teclas vistas en apartados anteriores.

Cuando el cursor ya esté donde nosotros queremos, podremos introducir alguna de las siguientes instrucciones; todas situarán a `vi` en modo edición, y el editor estará en disposición de recibir texto hasta que se pulse <`ESC`>:

- **i**: Escribe texto antes de la posición actual del cursor.
- **a**: Escribe texto después de la posición actual del cursor.
- **s**: Sustituye el carácter situado en la posición del cursor por todos los que se tecleen a continuación.
- **o**: Abre una línea por debajo de la actual y sitúa al editor en modo inserción.
- **O**: Abre una línea por encima de la actual y sitúa al editor en modo inserción.

Una vez que ya sabemos introducir texto, vamos a aprender a borrar partes del archivo editado. Para ello hemos de utilizar alguno de los siguientes mandatos:

- **dd**: Borra la línea actual completamente, sin importar la posición del cursor dentro de tal línea.

- **d<Intro>**: Borra la línea actual y la siguiente; al igual que con `dd`, no importa la posición del cursor dentro de la línea actual.
- **d<n_lineas>**: Borra *n_lineas* líneas a partir de la actual, sin contar a ésta (por tanto, son n+1 líneas).
- **dw**: Borra la palabra en la que se encuentra el cursor; si éste se encuentra en la mitad de una palabra, borrará desde ese punto hasta el final de la palabra.
- **x**: Borra el carácter sobre el que se encuentra el cursor.
- **d$**: Borra desde la posición actual de cursor hasta el final de la línea.
- **D**: Como `d$`, borra desde donde está el cursor hasta el final de la línea.
- **d0**: Borra desde la posición del cursor hasta el principio de la línea actual.

Si lo que queremos no es introducir texto, sino sustituirlo, podemos hacerlo utilizando uno de los siguientes mandatos:

- **r**: Reemplaza el carácter en el que está situado el cursor por el siguiente carácter pulsado.
- **R**: Reemplaza tantos caracteres como teclas se pulse; para finalizar la sustitución, hemos de pulsar <`ESC`> y volver a modo comando. Es similar al uso de la tecla <*Insert*> en un PC.
- **S**: Sustituye una línea del fichero por las que se escriban a continuación, comenzando la sustitución por la línea actual. Su ejecución finalizará también volviendo a modo comando.
- **cw**: Sustituye una palabra por el texto que se introduzca a continuación. Para finalizar hemos de pulsar <`ESC`>.
- **C**: Sustituye el texto desde la posición actual del cursor hasta el final de la línea actual. Finaliza con <`ESC`>.
- **cc**: Sustituye completamente la línea actual. Como todas las órdenes de sustitución, finaliza volviendo al modo comando del editor.

5.5. Otras órdenes de `vi`

Si nos hemos equivocado al ejecutar un mandato y necesitamos deshacer los cambios realizados, tenemos a nuestra disposición dos instrucciones:

- **u**: Deshace sólo el cambio realizado por el último mandato, aunque afecte a muchas líneas.
- **U**: Deshace varios cambios sobre una misma línea, aunque no sea la última modificación efectuada.

Si pulsamos la `u` dos veces, lo que estamos haciendo es deshacer el último cambio realizado, y luego deshacer éste otro cambio: en una palabra, volvemos a la situación original. Esto no sucede con `U`; una vez que se han desecho los cambios, pulsar `U` o `u` de nuevo no altera nada.

Otra instrucción de `vi` que nos va a resultar bastante útil va a ser **J**, que simplemente va a unir la línea inferior con la actual.

Si lo que deseamos es buscar dentro del fichero una determinada cadena de caracteres, pulsaremos (siempre en modo comando) la tecla `/`, introduciendo a continuación la cadena buscada. Al pulsar la barra `/`, notaremos que el cursor va hasta el final de la pantalla y queda esperando la introducción de una cadena; cuando encuentre el texto deseado, el cursor se moverá a la posición donde éste se encuentra por primera vez.

Si tras encontrar una determinada cadena queremos seguir buscando a lo largo del texto (`vi` nos indicará la primera que encuentre, no el resto si no le indicamos lo contrario), pulsaremos la tecla **n** o simplemente volveremos a pulsar `/`, esta vez sin necesidad de introducir de nuevo el texto a buscar.

Si queremos buscar en sentido inverso (desde la posición actual del cursor hacia el principio del archivo), en lugar de `/`, utilizaremos la tecla `?`, de uso análogo a la primera.

5.6. Órdenes orientadas a líneas

Todos las instrucciones orientadas a líneas van a comenzar con dos puntos (:). Si pulsamos esta tecla, el cursor se moverá a la parte inferior de la pantalla y quedará a la espera de que nosotros introduzcamos una orden.

Podemos ejecutar instrucciones del sistema operativo de esta forma; el formato será

`:!<orden>`

Si lo que queremos es salir unos instantes al *shell*, ejecutaremos un intérprete de órdenes de Unix disponible en el sistema, como

`:!bash`

De esta forma estamos en el *shell* para poder trabajar. Si queremos volver a `vi`, lo haremos con el mandato `exit`, visto ya.

Otra opción que se nos ofrece con las órdenes orientadas a líneas es la posibilidad de modificar las variables de entorno de `vi`, con la opción `:set`. Una de estas variables es la numeración de todas las líneas del archivo editado. Esto lo realizaremos con la orden

`:set number`

Los números de línea no se grabarán como texto, por lo que si volviéramos a ejecutar *vi* no nos aparecerían directamente; habríamos de ordenar de nuevo al editor que los indicase.
Para quitar la numeración de líneas, utilizaremos la orden

`:set nonumber`

que volverá el formato del texto a su formato original.

Otra variable de entorno susceptible de ser modificada es la diferenciación entre mayúsculas y minúsculas a la hora de buscar una cadena de caracteres. Para ello, usaremos la orden

`:set ignorecase`

o también

`:set ic`

Si queremos volver a diferenciar mayúsculas de minúsculas, restableceremos la forma original de la variable tecleando

`:set noignorecase`

o

`:set noic`

Si lo que queremos es grabar el contenido del archivo modificado sin tener que salir del editor, elegiremos

`:w`

Si a esta orden le indicamos un nombre de fichero diferente al que habíamos editado, nos realizará una copia en el nuevo archivo. En el caso que este archivo ya exista, y deseemos sobreescribirlo, utilizaremos el símbolo `!` a continuación de la `w`:

```
:w texto2.txt
:w! texto2.txt
```

Las órdenes orientadas a líneas también nos ofrecen la posibilidad de insertar en el archivo editado el contenido de otro fichero; para ello utilizamos la instrucción

```
:r <fichero>
```

y el fichero indicado aparecerá insertado a partir de la posición actual del cursor.

6. UNIX Y REDES: APLICACIONES

6.1. Introducción

El principal uso del sistema Unix hoy en día se da en entornos de red, especialmente en la conocida por Internet. Una red no es más que un número, más o menos elevado, de ordenadores conectados entre sí; la distancia entre ellos puede ser desde metros (una *LAN, Local Area Network*) hasta miles de kilómetros (*WAN, Wide Area Network*, que sería el caso de Internet). Detrás de cada uno de estos ordenadores interconectados existe un determinado número, más o menos elevado, de usuarios con los que nos puede interesar intercambiar información de cualquier tipo.

Las principales fuentes de información que podemos encontrar varían desde páginas *web*, en las que un determinado usuario ha expuesto un tema concreto, generalmente de una forma muy gráfica, hasta servidores de *ftp* anónimo, que no son más que ordenadores que permiten un acceso para obtener software de todo tipo. A lo largo de este tema veremos la relación de Unix con todo este tipo de utilidades de intercambio de información.

También es otro punto muy interesante el estudio las herramientas que ofrece Unix para el intercambio de correo electrónico (*e-mail, electronic mail*) entre millones de personas alrededor de todo el mundo, de una forma rápida y efectiva.

(a) Utilidades.

6.2. `tin`

UseNet es una red de propósito general que provee de un sistema de bases de datos (*BBS, Bulletin Board System*) llamado *netnews*. Estas 'noticias' se organizan en *newsgroups* atendiendo al tema del que tratan. Para ello se utilizará un determinado prefijo. Aquí damos una relación de los más comunes para nosotros:

comp .... Computadoras e Informática
rec .... Recreación, hobbies, aficiones...
sci .... Ciencias
soc .... Sociedad
talk .... Discusiones
alt .... Temas alternativos
es .... Newsgroups españoles
UPV .... Noticias locales de UPV

Cada prefijo irá seguido de un tema que distinguirá a los grupos de noticias entre sí; por ejemplo, en el grupo *comp.security.unix* se tratan temas referentes a la seguridad de los sistemas Unix, en *es.alt.chistes* se cuentan chistes en castellano, y en *UPV.anuncis* se publican anuncios de interés para la gente de la Universidad...

6.3. Uso de `tin`

Para poder leer los grupos de noticias en nuestra máquina, necesitaremos conectar a un servidor de *news*, por lo que deberemos ejecutar la orden `tin` con la opción *-r* (`tin -r`) o, lo que es lo mismo, la orden ya implementada `rtin`. De esta forma, aparecerán en nuestra pantalla todos los grupos a los que tenemos acceso.

Una vez tengamos en la pantalla los nombres de los grupos, podemos empezar a leer y escribir en cualquiera de ellos. Existen grupos de pruebas, para familiarizarse con `rtin`, y uno de ellos lo vamos a utilizar nosotros para enviar mensajes y leerlos: su nombre es *UPV.test*. Para ir a cualquier grupo de los existentes, pulsaremos la tecla **'g'** (*go to*), y el programa nos pedirá el nombre del *newsgroup* que queremos visitar. Cuando el indicador esté sobre *UPV.test*, pulsaremos *Intro* o *Tab*

para introducirnos en tal grupo...

Ahora ya estamos dentro del grupo de noticias. Podemos leer cada mensaje de los que existan, simplemente pulsando *Intro* de nuevo, desplazarnos entre mensajes con los cursores, enviar nuestros propios mensajes, etc.

Cada uno de esos mensajes puede tener varias respuestas. Para ir leyéndolas, y una vez que estemos en el visor de noticias, debemos pulsar la barra espaciadora, que servirá también para leer una nueva pantalla dentro del mismo mensaje. Debemos recordar que, en cualquier punto de la ejecución de `rtin`, podemos obtener ayuda pulsando la letra **'h'** (minúscula).

Supongamos que uno de los mensajes leídos es de nuestro interés, y deseamos grabarlo; `rtin` nos permite varias formas de hacerlo. Una de estas formas es enviarlo por correo a nuestra dirección *e-mail*; para esto, deberemos pulsar la letra **'m'**, y posteriormente indicar al programa nuestra dirección. La segunda forma que veremos es *save* (con la letra **'s'**), que nos grabará el artículo en nuestro directorio `/News`. *save* nos pedirá tanto el nombre que queremos poner al artículo que vamos a grabar como el proceso que debe seguir el grabado del artículo: si es texto, la opción correcta será *'n'*; si es una imagen o un binario *uucodeado*, deberemos elegir *u*, etc.

Supongamos ahora que nosotros también queremos enviar un comentario a cualquier grupo. Para ello, una vez dentro del grupo deseado, deberemos pulsar la letra **'w'** (*write*), e introducir el *subject* de nuestro mensaje (como vimos al tratar la orden `mail`, el *subject* debe ser un indicador del contenido del artículo que vamos a escribir). Después de esto, entraremos en el editor de textos `vi` y podremos empezar a escribir, utilizando las opciones normales de tal editor. Para finalizar, saldremos de `vi` (por ejemplo con *ESC :wq*), y el sistema nos pedirá la confirmación para enviar el mensaje al grupo correspondiente.

Si lo que queremos hacer es responder a un mensaje enviado por otra persona, tenemos dos formas básicas de hacerlo. La primera es enviar nuestra respuesta directamente a la dirección *e-mail* del usuario que envió el mensaje al grupo, para lo que utilizaremos la opción **'r'** (*reply*), y podremos comenzar a escribir nuestra respuesta. La segunda es añadir el comentario que deseamos hacer en el mismo grupo en que está el original, y también en el mismo hilo de noticia. Para ello, utilizaremos la opción **'f'** de `rtin`, y podremos añadir nuestra respuesta en el *newsgroup* para que cualquiera pueda leerla.

Cuando hayamos concluido nuestra sesión con `rtin`, podemos volver a la pantalla anterior o al intérprete de órdenes simplemente pulsando **'q'** o **'Q'**.

6.4. `lynx`

En muchos ordenadores los usuarios tienen páginas WWW (*World Wide Web*) con información sobre infinidad de temas; a ellas podemos acceder mediante los denominados programas navegadores, como Netscape o Explorer, que nos mostrarán por pantalla la información consultada en modo gráfico.

Pero como nosotros sólo vamos a poder trabajar en modo texto, necesitamos otro tipo de navegador para consultar las páginas que deseemos; este programa se denomina `lynx`. Su sintaxis básica es *lynx <URL>*, donde *<URL>* indica la dirección de tal página. Esta URL generalmente irá precedida por *http://*, aunque no en todos los casos va a ser necesario. Así, para consultar la *web* de *andercheran.aiind.upv.es*, sólo hemos de teclear

```
rosita:~# lynx http://andercheran.aiind.upv.es
```

y acto seguido estaremos ya dispuestos a 'navegar' por las páginas del sistema y sus *links* (conexiones a otros recursos WWW).

6.5. Uso de `lynx`

Si tras indicarle a `lynx` una URL no es capaz de conectar a tal dirección, por cualquier motivo, nos lo indicará con un mensaje de alerta en la barra de estado de la parte inferior de la pantalla. Si no hay ningún problema, y la conexión es correcta, ya estaremos viendo la página indicada en la línea de órdenes. En este punto, podemos introducirle a `lynx` las opciones necesarias para que realice lo que nosotros deseemos. Veamos los mandatos que nos van a interesar para desenvolvernos con soltura entre los millones de páginas WWW de todo el mundo:

Los cursores nos van a permitir movernos por la página de arriba a abajo, y también acceder a un *link* (podemos identificar los enlaces porque estn 'sombreados' de color blanco) con el cursor de la derecha (→), o volver al *link* anterior con el de la izquierda (←).

Es posible que nos interese grabar, por cualquier motivo, la página que estamos viendo. `lynx` nos permite hacerlo de tres formas diferentes, todas accesibles pulsando la tecla **'p'**. Si lo hacemos, veremos las posibles opciones: grabar la página en un archivo, enviarla por correo a una dirección determinada (obviamente, la nuestra), o imprimir la página en pantalla (esta última opción sólo suele ser útil si estamos utilizando algún capturador de pantallas, como `script` - no visto en el temario -). Si elegimos grabar el contenido de la página, `lynx` nos preguntará el nombre que deseamos ponerle; por defecto, nos dará el que ésta ya tiene. Si nos interesa otro nombre, pues simplemente borramos el de defecto y tecleamos el deseado. Al elegir enviar por correo la página, se nos va a preguntar por la dirección deseada. Introduciremos nuestro *e-mail*, y en unos instantes tendremos la página en nuestro buzón.

Si estamos visitando una WWW, y sin salir de `lynx` deseamos cambiar a otra dirección, hemos de pulsar **'g'** (*Go*). El programa nos preguntará por la nueva URL que deseamos visitar. También es posible que después de seguir varios *links* deseemos volver al principio de nuestro viaje, a la página principal, para seguir otro camino diferente. Para ello podemos pulsar las veces necesarias el cursor de la izquierda, o, de una forma más cómoda y rápida, elegir la opción **'m'**.

Existen páginas muy extensas, y quizás no queramos perder el tiempo leyendo todo su contenido, sino ir directamente a una parte que nos pueda interesar; `lynx` nos permite buscar una determinada cadena de caracteres dentro de una página: pulsamos **'/'** y el programa nos preguntará por la cadena seleccionada. Así, si sabemos que en la página hay información sobre Unix, por ejemplo, nos bastará indicar la cadena *unix* , para que automáticamente nos lleve a la parte donde están los datos buscados.

Acabamos de decir que después de estar un rato siguiendo *links* a través de la red es posible que andemos algo perdidos con relación al lugar donde nos encontramos en estos momentos; pulsando *backspace*, `lynx` nos mostrará el camino que hemos seguido hasta este instante.
Finalmente, cuando hayamos concluido nuestro viaje, pulsaremos **'q'** para abandonar `lynx` y volver al *prompt* de nuestro sistema.

6.6. `gopher`

El *gopher* de Internet es un servicio de intercambio de información que permite a sus usuarios explorar, buscar y recibir documentos situados en diferentes servidores de todo el mundo. Esta información aparece frente al usuario como un sistema de archivos y directorios, que aunque pueden encontrarse físicamente a miles de kilómetros del servidor al que hemos conectado, nos va a parecer que viene del mismo lugar.

El tipo de información que podemos encontrar en un servidor *gopher* varía desde archivos de texto hasta binarios, imágenes, sonidos... Los *links* que tiene un servidor hacia otros constituyen el denominado *gopherspace*, una red de intercambio de información alrededor de todo el mundo. En muchos

aspectos, *gopher* es equiparable al modo de intercambio de información utilizado por WWW, pero en el *gopherspace* no vamos a disponer de información en modo gráfico como nos permitirían tener los servidores WWW.

Desde nuestro entorno Unix vamos ser capaces de acceder a servidores *gopher* públicos, utilizando para ello el mandato `gopher`, al que indicaremos la dirección del server que nos interese: *gopher <direccion>*. Una vez hayamos conectado a un servidor, veremos que el menú que se nos presenta hace muy fácil moverse por el *gopherspace*, recibir ficheros, visualizar documentos, etc. Por tanto, y también porque ya conocemos otro método de acceder a servidores de información WWW, que ya hemos comentado que son en muchos puntos similares a los servidores *gopher*, no vamos a entrar con detalle en el manejo de *gopher*; como ejemplo de su facilidad de manejo, basta decir que casi todo lo que deseemos hacer será posible utilizando los cursores e *Intro*.

En la actualidad el *gopherspace* está desapareciendo (si no lo ha hecho ya), y la navegación por el mismo ha sido sustituida por la navegación *web*, mucho más gráfica. No obstante, y aunque por este motivo no vamos a entrar en el manejo de la herramienta (sería difícil hoy en día encontrar incluso servidores *gopher* operativos), siempre es aconsejable conocer que un día existió, y que la *web* no ha estado ahí desde siempre…

6.7. `ftp`

FTP significa *File Transfer Protocol*, y como su nombre indica no es más que un protocolo para transferencia de ficheros implementado para casi todos los sistemas operativos, y por supuesto también para Unix.

En un determinado momento nos puede interesar un archivo que está en una máquina remota, o simplemente copiar en disquetes el contenido de nuestro directorio en la máquina Unix donde trabajamos; para ambas cosas habremos de utilizar `ftp`.

La sintaxis de la orden `ftp` es sencilla: *ftp <direccion>*. Después de esto, `ftp` nos pedirá un *login* y un *password* para acceder al sistema deseado, y si ambos son correctos ya estaremos en disposición de transferir ficheros de un lugar a otro.

Si lo que queremos es copiar en un disco archivos que están en la máquina remota donde trabajamos, desde el PC, y bajo MS-DOS, teclearemos

```
C:\> ftp servidor.upv.es
```

introduciremos nuestro nombre de usuario y nuestra clave, y podremos empezar a trabajar con los archivos.

Si por el contrario estamos conectados a una máquina Unix, y deseamos transferir ficheros desde otro sistema, el mecanismo a seguir será por lo general algo distinto; existen infinidad de servidores FTP que permiten un acceso público para conseguir determinados archivos. Este tipo de ordenadores se denominan de *ftp* anónimo, ya que cualquiera puede acceder a ellos para transferir ficheros entre las dos máquinas. En este caso, como el acceso es público pero está limitado a unos ciertos archivos, ya que no somos usuarios de ese sistema, nuestro *login* será *anonymous* o *ftp* por lo general, y como clave introduciremos nuestra dirección de correo electrónico.

En ambos casos disponemos de las mismas instrucciones básicas para tratar los archivos, aunque éstos dependen del cliente *ftp* utilizado; veamos cuáles son:

- *!*: Suspende la sesión *ftp* y vuelve al intérprete de órdenes de nuestro sistema.
- *ASCII*: Establece un modo ascii de transferencia de archivos; este modo será el que utilizaremos para trabajar con ficheros de texto.

- *BELL*: Activando esta opción, escucharemos un pitido cada vez que se finalice la ejecución de un mandato.
- *BINARY*: Establece, contrariamente a *ASCII*, un modo binario de transferencia de archivos; lo usaremos para trabajar con archivos binarios principalmente, pero es lo habitual para transferir archivos de cualquier tipo
- *BYE/QUIT/CLOSE/DISCONNECT*: Finaliza la sesión *ftp* y volvemos al intérprete de órdenes de nuestro sistema. No hemos de confundir con la orden *'!'*, que sólo suspende la sesión para retornar posteriormente a ella tecleando `exit` en el *prompt*.
- *CD*: Cambiamos de directorio en la máquina a la que hayamos conectado mediante *ftp*.
- *LS/DIR*: Lista los archivos del directorio actual del sistema al que hemos conectado.
- *GET*: Sirve para recibir un archivo desde el servidor *ftp* hasta nuestro sistema. Antes de transferir ficheros, es conveniente establecer el modo correcto (ascii o binario) para evitar errores en la transmisión.
- *HASH*: Imprime en pantalla el símbolo *'#'* por cada Kbyte transferido de un sistema a otro.
- *?/HELP*: Nos ofrece ayuda de los mandatos disponibles en el servidor al que hayamos conectado; si tecleamos *help <orden>*, nos ofrecerá ayuda acerca de la orden específica que indiquemos.
- *LCD*: Cambiamos el directorio local, esto es, el directorio de la máquina desde la que hemos iniciado la sesión, no del servidor.
- *MGET*: Utilizaremos `mget` para recibir múltiples archivos. Si la opción *prompt*, que veremos más adelante, está activa, nos preguntará que confirmemos la transferencia de cada uno de los ficheros.
- *MPUT*: Contrario a `mget`, sirve para enviar archivos al sistema al que hayamos conectado mediante *ftp*; generalmente, si es un servidor de *ftp* anónimo, sólo podremos escribir en un directorio llamado `/incoming`.
- *OPEN*: Si en el *prompt* de nuestro sistema hemos tecleado solamente *ftp*, sin indicar una dirección, la instrucción `open` nos servirá para abrir una conexión con un servidor; su sintaxis es sencilla: *open <direccion>*.
- *PROMPT*: Activa o desactiva la confirmación para los mandatos que trabajan con múltiples archivos, como `mput` o `mget`.
- *PUT*: Contrario a `get`, envía un archivo desde nuestro sistema hasta el servidor. Como antes, hemos de tener cuidado con el modo de transferencia que vayamos a utilizar.
- *PWD*: Imprime el directorio actual del servidor.
- *REGET*: Si nos hemos quedado a medias al intentar recibir un archivo, no es necesario volver a empezar la recepción: `reget` nos permite continuar desde el punto en que se perdió la conexión.
- *RSTATUS*: Si hemos establecido una conexión, `rstatus` nos indicará nuestras variables de estado (modo ascii/binario, *prompt* on/off...) en el servidor.
- *SIZE*: Nos da el tamaño de un archivo en el servidor.
- *STATUS*: Nos indicará nuestro estado en la máquina local, hayamos o no conectado a un servidor.

- *USER*: Si durante una sesión *ftp* queremos cambiar nuestro nombre de usuario en el servidor sin salir de él, teclearemos *user <nuevo_nombre>*.

Para finalizar el estudio de *ftp*, cabe destacar que su uso se está limitando cada día más a procesos lanzados con `nohup` (sin control de terminal), es decir, sin el usuario conectado físicamente. Esto es debido a que con `lynx` podemos realizar también transferencia de ficheros en muchos servidores de ftp anónimo, indicando *ftp://* al principio de lo que sería la URL.

6.8. `telnet`

`telnet` nos va a permitir conectar con un servidor remoto para iniciar una sesión de trabajo en el sistema Unix; ya hemos utilizado `telnet` para acceder al ordenador *servidor.upv.es*, donde estamos realizando el curso, desde nuestro PC. Una vez dentro del sistema Unix, puede ser que nos interese ejecutar de nuevo `telnet` para acceder a otro sistema.

Una vez realizada una conexión, `telnet` actúa como un intermediario entre nosotros y el ordenador al que hemos llamado; cuando pulsemos una tecla, ésta se enviará al sistema remoto, y cada vez que este sistema remoto produzca una respuesta se envía a nuestra terminal. De esta forma, nos va a parecer que nuestro teclado y monitor están conectados directamente al *host* que hemos llamado.

Si ejecutamos la orden sin indicar ninguna dirección de máquina, entraremos en el modo comando de `telnet`; entonces podremos empezar a comunicarnos con el sistema introduciendo cualquiera de las diferentes opciones. Veamos las que nos van a ser más útiles:

- *C/CLOSE*: Cierra una conexión que hayamos establecido previamente con un sistema.
- *L/LOGOUT*: Desconecta del sistema remoto y cierra completamente la conexión.
- *O/OPEN <DIRECCION>*: Conecta a la dirección especificada. Hemos de recordar que necesitamos un *login* y una clave para acceder al sistema que hayamos indicado.
- *Q/QUIT*: Sale del modo comando de `telnet` y vuelve al *prompt* del sistema en el que hemos ejecutado la orden.
- *Z*: Suspende momentáneamente la ejecución de `telnet` y retorna al sistema; para volver a *telnet*, podemos teclear `fg` (*foreground*, cuyo uso veremos más tarde).
- *H/HELP/?*: Nos da una ayuda de los diferentes mandatos que admite la orden `telnet`.

6.9. `finger`

El servicio *finger* permite conseguir información acerca de casi cualquier persona conectada a Internet. Decimos 'casi', porque si al administrador de nuestro sistema no le interesa que alguien vea nuestra información no va a permitir que la máquina en que estamos trabajando tenga un servicio *finger* de cara al exterior, tanto por motivos de seguridad como para evitar tráfico en la red.

Vamos a ver aquí diferentes formas de utilizar *finger* para conocer datos de un sistema en general o de un usuario de ese sistema, saber lo que significa la información proporcionada, y también utilizar nuestra propia información de cara al resto de usuarios.

Si ejecutamos `finger` en nuestro *prompt*, sin pasarle ningún argumento, nos va a informar de los usuarios conectados al servidor en esos momentos; podemos conocer información más específica de un usuario determinado indicándole a `finger` el identificador (*login*) de tal usuario:

```
rosita:~# finger
Login  Name             Tty  Idle When Where
```

```
root El Spiritu Santo      1         03:23
toni    Toni Villalon         2         01:23
toni Toni Villalon        p0    10   01:45 localhost
rosita:~# finger toni
Login name: toni In real life: Toni Villalon
Phone:
Directory: /home/toni Shell: /bin/bash
Last login Wed Oct 30 on ttyp0
No unread mail
No plan
rosita:~#
```

Vemos como al indicar un nombre de usuario la información proporcionada por `finger` es más precisa que si no lo hacemos. Se nos va a indicar el nombre verdadero del usuario, el teléfono, el directorio *$HOME*, el intérprete de órdenes que utiliza por defecto, cuándo fue la última vez que se conectó, si tiene o no correo, y cual es su *plan* (hablaremos más adelante sobre los ficheros `.plan` y `.project`). Esta información en manos de un potencial intruso podría ser comprometedora, por lo que, como hemos comentado anteriormente, el servicio *finger* no siempre va a estar disponible en un sistema.

Hasta ahora hemos visto la utilidad de `finger` dentro de nuestro propio sistema; también podemos conocer datos de otro ordenador que posea este servicio, simplemente indicando como argumento a `finger` la dirección IP o el nombre del sistema que deseemos analizar, precedido por el símbolo de la arroba (*'@'*), o la dirección de correo de un usuario, si es que queremos conocer algo sobre él. Veamos un sencillo ejemplo:

```
rosita:~# finger @marilyn
Login          Name              Tty            Idle   When    Where
root   El Spiritu Santo          1              03:23
toni   Toni Villalon             2              01:23
toni   Toni Villalon             p0             10     01:45   localhost

rosita:~# finger toni@marilyn
Login name: toni                       In real life: Toni Villalon
Phone:
Directory: /home/toni                  Shell: /bin/bash
Last login Wed Oct 30 on ttyp0
No unread mail
No plan
```

Podemos ver que la salida es análoga a cuando hacíamos un *finger* dentro de nuestro sistema. La única diferencia es que hemos de indicar con la arroba el sistema destinatario de nuestro *finger*. Si no lo hacemos de esta manera, *finger* pensará que estamos refiriéndonos a un usuario de nuestro ordenador, y nos dará una salida parecida a la siguiente:

```
rosita:~# finger pleione.cc.upv.es
Login name: pleione.cc.upv.es          In real life: ???
rosita:~#
```

(b) Gestores de correo.

6.10. `elm`

`elm` es un gestor de correo interactivo disponible en la mayoría de clones de Unix, por supuesto también en Linux. Está orientado a pantalla, y por su potencia ha superado como gestor usual tanto a `mail` como a `mailx`.

Aunque existen varias formas de invocar a `elm`, vamos a ver la más común: simplemente habremos de teclear el nombre del gestor en nuestro *prompt*. De esta manera, entraremos en el programa y podremos comenzar a gestionar nuestro correo, tanto el que hemos recibido como el que vamos a enviar.

La primera vez que ejecutemos `elm`, se van a crear en nuestro directorio *$HOME* un par de subdirectorios, `Mail/` y `.elm/`, que van a ser necesarios para almacenar nuestro correo y para mantener una configuración personalizada del gestor (por ejemplo, definiendo cabeceras para los mensajes o *aliases* de alguna dirección con la que nos comuniquemos a menudo).

Si al entrar en el entorno que nos proporciona `elm` tenemos correo en nuestro buzón electrónico (un archivo con nuestro identificador de usuario situado en el directorio `/var/spool/mail/`), aparecerán ante nosotros los *subject* de los mensajes junto con su remitente, marcando el actual con un 'sombreado' blanco; esta marca la podremos desplazar entre los mensajes, utilizando para ello los cursores y también los caracteres **'+'** (aparece la siguiente página del índice de mensajes), **'-'** (aparece la página anterior a la actual), **'J'** (avanza al siguiente mensaje) o **'K'** (retrocede al mensaje anterior al actual); si introducimos un número y luego pulsamos *Intro*, el mensaje actual será el marcado con el número pulsado.

Podremos operar siempre sobre el mensaje que tengamos marcado como actual en ese momento. La primera acción que nos puede interesar realizar sobre él seguramente será leer su contenido (denominado *body* o cuerpo del mensaje). Para ello pulsaremos *Intro* o la barra espaciadora, y el mensaje aparecerá paginado en nuestra terminal.

Una vez hemos leído un mensaje, podemos responder a su remitente. Esto lo conseguimos mediante la tecla **'r'**. `elm` nos preguntará si queremos copiar el original en la respuesta; contestar **'y'** nos va a ser útil para recordar a quien la lea la respuesta el origen de nuestros comentarios.

De esta forma vamos a ir trabajando: moveremos el indicador del mensaje actual para ir leyendo todo el contenido de nuestro buzón, respondiendo mensajes, etc. Cuando hayamos leído un mensaje, es posible que nos interese grabarlo en un archivo donde podemos ir almacenando los mensajes referentes a un mismo tema. Este archivo se llama *folder*, y para grabar el mensaje en un *folder* determinado, elegiremos la opción **'s'**, tras la cual `elm` nos preguntará por el nombre del archivo donde deseamos grabarlo; otra forma de hacer esto es con la opción '>' de `elm`, a la que indicaremos el *folder* adecuado. Si más tarde deseamos visualizar los archivos grabados en un fichero, habremos de abrirlo. `elm` siempre abre por defecto nuestro buzón, y le hemos de indicar si queremos leer de otro lugar con la opción **'c'**, para cambiar de *folder* (los *folders* estarán almacenados en nuestro directorio *$HOME/Mail*).

Es posible que aparte de responder a un mensaje deseemos enviar una copia del original a algún amigo. Esto se llama enviar un *forward*, y lo podremos realizar con la opción **'f'**; incluso podremos modificar algo del mensaje antes de enviarlo.

También necesitaremos borrar mensajes que no nos interese conservar. Ello se consigue simplemente pulsando **'d'**, lo que marcará el mensaje actual como borrado. Antes de abandonar `elm`, con la opción **'q'**, el programa nos preguntará si deseamos borrar los mensajes marcados (que tendrán una *'D'* al lado del nombre del remitente). Si después de marcarlo, deseamos conservarlo, **'u'** realiza un *undelete* del mensaje actual.

Si lo que nosotros deseamos no es gestionar el correo recibido sino enviar el nuestro propio, debemos elegir la opción **m** de `elm`. Entonces, el programa nos preguntará los datos del mensaje a enviar (destinatario, tema...), tras lo cual entraremos en un editor de texto (en nuestro caso será `vi`), en el que teclearemos nuestro mensaje.

Para finalizar con el uso del gestor de correo `elm`, hemos de recordar que en todo momento podemos conseguir ayuda acerca de las opciones disponibles pulsando la tecla '`?`', y que podremos redefinir algunas variables de usuario a nuestro gusto tecleando '`o`'.

6.11. pine

Más que como un gestor de correo, `pine` se suele definir a menudo como un gestor de mensajes, ya que su configuración permite tanto trabajar con el *e-mail* habitual como con las *news* de Usenet. Sin embargo, como nosotros ya sabemos utilizar el lector de noticias `tin`, sólo vamos a aplicar `pine` para gestionar, al igual que hemos hecho con `elm`, nuestro correo electrónico.

`pine` se puede configurar de muchas formas, atendiendo principalmente al conocimiento que el usuario tenga de su uso; para nosotros, las opciones elementales van a ser las mismas que nos ofrecía `elm`, aunque dentro de un entorno mucho más amigable que éste.

En primer lugar, vamos a ver cuáles son las utilidades a las que podemos acceder desde el menú principal, el que se nos muestra en pantalla inmediatamente después de ejecutar `pine`:

- `HELP`: Nos muestra el fichero de ayuda del gestor.
- `COMPOSE MESSAGE`: Se utiliza para editar y enviar un mensaje a una determinada dirección.
- `FOLDER INDEX`: Nos permite acceder a los mensajes del *folder* actual.
- `FOLDER LIST`: Mediante este servicio vamos a poder seleccionar un determinado *folder* (almacenados en `$HOME/mail`) o nuestro buzón, que se llamará `INBOX` por defecto (`/var/spool/mail/`<`usuario`>).
- `ADDRESS BOOK`: Aunque nosotros no vamos a utilizar una lista de direcciones, esta opción nos permite crear una especie de agenda con direcciones de correo que utilicemos frecuentemente.
- `SETUP`: Nos permite definir un entorno personalizado para `pine`: editor a utilizar, nombre que deseamos que aparezca como remitente cuando enviemos *mail*, etc.
- `QUIT`: Sale de `pine`.

A cada una de estas opciones podremos acceder pulsando las teclas '`?`', '`C`', '`I`', '`L`', '`A`', '`S`' o '`Q`', respectivamente, desde casi cualquier menú de `pine`.

Para nosotros, la opción más interesante y la que más vamos a utilizar va a ser *Folder List (L)*, ya que nos va a permitir seleccionar un *folder* y trabajar con los mensajes contenidos en él; si elegimos esta opción, veremos en pantalla una lista con todas las carpetas de correo que tenemos. Tras seleccionar una determinada, veremos el remitente y el tema de todos los mensajes grabados en el archivo.

En este punto podemos comenzar a realizar acciones sobre un determinado mensaje. Si elegimos uno como actual (con los cursores, o las teclas '`-`', barra espaciadora, '`p`' o '`n`', que nos desplazan de una a otra ventana o mensaje), podremos leerlo (*Intro* o '`v`'), realizar un *forward* ('`f`'), responder al remitente ('`r`'), marcarlo como borrado ('`d`'), desmarcarlo ('`u`') o grabarlo en un *folder* ('`s`'). Esta última opción automáticamente marcará el mensaje como borrado, para evitar tener el mismo texto en varios lugares a la vez; de cualquier forma, antes de abandonar `pine` ('`q`'), el programa nos preguntará si realmente queremos borrar los mensajes marcados. Cuando terminemos de trabajar con los mensajes de un *folder*, podemos tanto volver al menú principal ('`m`') como a la lista de *folders* ('`l`').

No vamos a ahondar más en el uso de `pine`, ya que este gestor puede llegar a admitir un uso muy complicado, y además en cada pantalla el programa nos informa de las opciones que tenemos disponibles, generalmente pulsando '`?`'.

7. CONCEPTOS DEL SISTEMA OPERATIVO UNIX

7.1. Ficheros

Lógicamente, un archivo es un conjunto de datos relacionados de alguna forma entre sí; físicamente, para el sistema Unix, un archivo no va a ser más que un conjunto de *bytes* almacenados en un disco del sistema o en cualquier otro medio de almacenamiento secundario.

En nuestra máquina Linux vamos a encontrar tres tipos básicos de ficheros: ficheros planos, directorios, y ficheros especiales. Un **fichero plano** es aquél que contiene información generada durante una sesión de trabajo de cualquier usuario, ya sean programas, documentos, ejecutables... Un **directorio** es una especie de catálogo de archivos planos manipulado por el S.O. para presentar una estructura arborescente del sistema de archivos. Para nosotros, un directorio va a ser un fichero especial que va a contener archivos de todo tipo: planos, otros directorios (llamados subdirectorios) o ficheros especiales. Siempre existirán dos archivos dentro de cualquier subdirectorio, denominados `'.'` y `'..'`, que hacen referencia al propio directorio (`'.'`) y a su directorio padre (`'..'`).

Los ficheros especiales pueden ser de varios tipos; en primer lugar, encontramos los *archivos de dispositivos*. Para Unix, cualquier parte del computador (disco, cinta, tarjeta de sonido, impresora, memoria...) es un archivo especial. Existen dos tipos básicos de archivos de dispositivo: **orientados a carácter** (que realizan sus operaciones de I/O carácter a carácter, como impresoras o ratones) y **orientados a bloque** (realizan esas mismas operaciones en bloques de caracteres; los más comunes son los discos y las cintas).

Otro tipo de archivo especial es el **link** o enlace, que no es más que una copia de un archivo determinado en otro lugar del almacenamiento, una especie de segundo nombre del archivo referenciado. Podremos distinguir dos tipos de enlaces: los duros y los simbólicos; mientras que los primeros realizan una segunda copia del archivo enlazado, los segundos son simplemente apuntadores a ese archivo.

Por último existen dos tipos de archivos especiales que no vamos a tratar con detalle, ya que suelen ser sólo usados en programación o por el propio sistema operativo: son los **sockets** y los **pipes** o *FIFOs* (*First In First Out*). Los primeros son una especie de 'agujeros negros' que reciben y dan información, mientras que los segundos son las denominadas 'tuberías' del sistema: los datos que entran por un extremo salen en el mismo orden por el extremo contrario.

Podremos distinguir entre sí los diferentes archivos a partir de su primer bit de descripción (que en realidad no es un solo bit); los ficheros planos se denotarán por `'-'`, los directorios por `'d'`, los dispositivos orientados a carácter por `'c'`, los orientados a bloque por `'b'`, los *pipes* por `'p'`, y los *sockets* por `'s'`. Veamos un ejemplo, al listar el contenido de un directorio con `ls -al`:

```
rosita:~$ ls -al
Estos directorios son el actual y el padre:
drwxr-xr-x 15 toni users 2048 Nov 2 03:49 ./
drwxr-xr-x 5 root root 1024 Jun 9 02:45 ../
Directorios normales: vemos la 'd' al principio de los permisos:
drwxr-xr-x 2 toni users 1024 Oct 26 15:18 .pgp/
drwxr-xr-x 2 toni users 1024 Oct 26 17:11 programacio/
Archivos planos: al principio, tenemos '-':
-rw-r-xr-x 1 toni users 155 Nov 2 03:49 test
-rwxr-xr-x 1 toni users 349 Oct 3 05:14 server
Pipes: vemos su indicador, 'p':
prw-r-xr-x 1 toni users 155 Nov 2 03:49 .plan
prwxr-xr-x 1 toni users 369 Oct 3 12:12 pipetest
Archivos especiales orientados a carácter o a bloque:
```

```
crw-r-xr-x 1 toni users 0 Nov 2 06:29 caractest
brwxr-xr-x 1 toni users 0 Oct 2 08:14 blocktest
```

Enlaces simbólicos; su indicador es la 'l':

```
lrwxrwxrwx 1 toni users 10 Oct 2 22:25 magic ->/etc/magic
lrwxrwxrwx 1 toni users 11 Oct 2 22:25 passwd ->/etc/passwd
```

7.2. Permisos de los archivos

En este apartado vamos a intentar explicar qué es el modo de un fichero, cómo se interpreta y cómo se modifica (aunque esto último lo vimos en el capítulo 3, al estudiar la orden `chmod`).
En Unix existen tres tipos de acceso a un fichero:

- Lectura (*Read*): Permite visualizar el contenido de un archivo.
- Escritura (*Write*): Permite modificar o borrar tal contenido.
- Ejecución (*eXec*): Permite ejecutar un archivo (binario o proceso por lotes); sobre un directorio, permite utilizar su nombre como parte del camino de un archivo.

Dentro de un mismo fichero, existen tres niveles para los tipos de acceso anteriores:

- Dueño del fichero (*User*): Aplicable al creador del archivo.
- Grupo del fichero (*Group*): Correspondiente al grupo al que pertenecía el dueño en el momento de crear el archivo.
- Otros (*Others*): Resto de usuarios.

De esta forma, los permisos sobre un archivo van a componerse de tres ternas, nueve componentes que corresponden a los tipos de acceso y a sus niveles:

r w x   r w x   r w x
Terna 3: Correspondiente al nivel *others*
Terna 2: Correspondiente al nivel *group*
Terna 1: Correspondiente al nivel de dueño

Cada uno de los elementos de la terna es el tipo de acceso definido para el nivel determinado (*Read, Write, eXec*). Si alguno de los permisos no está activo, aparecerá un guión (-) en su lugar (pe., *rwxr-xr-x*).

El modo de un archivo puede representarse mediante una máscara binaria, con un 1 si está activo, o un 0 si no lo está; a su vez, cada tres bits pueden representarse por su dígito octal, por ejemplo para cambiar el modo de un fichero. Veamos un ejemplo:

rwx r-x r-x

En binario, tendríamos una máscara para este modo de archivo que sería

111 101 101

Si pasamos este número binario al sistema de numeración octal, tenemos

7 5 5

Luego el modo del archivo es el 755: su dueño puede leerlo, modificarlo y ejecutarlo; los usuarios pertenecientes a su grupo pueden leerlo y ejecutarlo, al igual que el resto de usuarios. Si el nombre del fichero fuera `file.txt`, por ejemplo, es posible que su dueño hubiera realizado un cambio de su modo hasta conseguir el 755:

```
rosita:~# chmod 755 file.txt
```

Podríamos cambiar el modo del archivo para conseguir que nadie lo pueda leer, para que sólo lo puedan ejecutar los miembros de nuestro grupo, etc. Veremos ejemplos en clase para estudiar el modo de un fichero y las posibilidades que ofrece tanto a su dueño como al resto de usuarios de un sistema Unix.

7.3. Archivos ejecutables, imágenes y procesos

Un archivo ejecutable es simplemente un programa que puede procesarse en la computadora. En Unix encontramos dos tipos de ejecutables: *shellscripts*, que no son más que archivos en los que hemos especificado una serie de instrucciones del sistema para que se ejecuten en orden, y binarios, que son ficheros planos con código máquina interpretado directamente por el procesador (generados a partir de un código fuente, un compilador y un montador o *linker*).

Cuando ejecutamos cualquier archivo capaz de ser procesado por el ordenador, se carga en memoria principal una imagen de ese fichero, que no es más que una copia del archivo o de un trozo de éste. En realidad, una imagen también está compuesta por el entorno que proporciona el sistema para llegar a ejecutar el archivo, pero éste es un tema demasiado complejo para abordarlo aquí.

Hemos explicado ya que Unix es un S.O. multitarea y multiusuario. Normalmente, en la máquina estarán conectados varios usuarios, cada uno trabajando con una serie de ejecutables, y también existirán trabajos del sistema, imágenes en memoria susceptibles de ser ejecutadas. Por tanto, la CPU ha de cambiar constantemente entre tareas, ejecutando una porción de cada una durante uno o varios ciclos de reloj. De esta forma, ante nosotros parecerá que nada más introducir un mandato el sistema directamente nos va a dar una respuesta, aunque en realidad lo que está haciendo es ejecutar varias tareas a la vez, cambiando rápidamente entre ellas, para poder dar a todos los usuarios la misma sensación de dedicación plena. Cuando la CPU ejecuta un trozo de imagen, esta imagen se convierte en un proceso. Como en un intervalo de tiempo más o menos reducido se van a ejecutar todas las imágenes que coexisten en el sistema, por extensión del lenguaje se suele confundir el concepto de proceso con el de imagen (por ejemplo, la orden `ps` no debería ser *Process Status* sino *Image Status*).

7.4. El *shell*

El *shell* no es más que el intérprete de órdenes, un programa del sistema que nos va a permitir comunicarnos con la máquina ordenándole cosas.

Linux, como cualquier Unix, nos va a ofrecer distintos procesadores de órdenes, aunque nosotros sólo vamos a comentar las posibilidades de los más comunes: *Bourne Shell* (`sh`), *Bourne Again Shell* (`bash`) y *Korn Shell* (`ksh`). *C Shell*, también muy popular, no lo vamos a tratar, ya que su uso es más adecuado para los programadores de Unix. Una tabla resumen de los distintos *shells* podría ser la siguiente:

Programa	*Intérprete*
sh	*Bourne Shell*
tcsh	*C Advanced Shell*
ash	*Shell reducido*
zsh	*Z Shell*
bash	*Bourne Again Shell*
csh	*C Shell*
ksh	*Korn Shell*
pdksh	*Public Domain Korn Shell*

Bourne Shell es el intérprete de órdenes fundamental en todo sistema Unix, ofreciéndose siempre de serie. Sabremos que lo estamos utilizando porque su indicador, su *prompt*, será el símbolo `'$'`,

aunque este símbolo corresponde también a *Bourne Again Shell*, y además es fácilmente modificable por el usuario.

Korn Shell es una extensión de *Bourne Shell*, que mejora a éste último. Todo lo que funcione con `sh` lo hará también con `ksh`. Sus prestaciones son similares a *C Shell*, incorporando histórico de órdenes, control de tareas y creación de *aliases*.

Bourne Again Shell es el intérprete de *Free Software Foundation*, una asociación con fines no lucrativos cuyo objetivo es desarrollar y suministrar a todo el mundo sistemas Unix gratuitos. Amplía las posibilidades de *Bourne Shell*, incorporando mejoras como el rodillo de órdenes (posibilidad de repetir cualquier instrucción anterior utilizando el cursor hacia arriba). Dadas sus excelentes características, va a ser el *shell* que nuestra máquina Linux (no así otros Unices) nos va a proporcionar por defecto.

7.5. Programación en *shell*

Como ya hemos comentado, el *shell* de Unix es un intérprete de órdenes. Como cualquier intérprete o traductor, define un lenguaje de programación que tiene unas características como:

- Variables.
- Metacaracteres (palabras y caracteres reservados).
- Procedimientos (*shellscripts*).
- Estructuras de control de flujo, como *if* o *while*.
- Manejador de interrupciones.

Aquí no vamos a intentar profundizar mucho en la programación utilizando el *shell*, ya que sería imposible en tan sólo unas horas. Sin embargo, sí que vamos a pretender conocer mínimamente esta programación y poder interpretar programas sencillos.

Las órdenes que ha de ejecutar el *shell* pueden ser leídas desde el teclado o desde un fichero (*shellscript*). Las líneas de este fichero que comienzan por el símbolo '`#`' son interpretadas como comentarios. El formato de un *shellscript* no difiere mucho a la secuencia de órdenes que habríamos de introducir por teclado; veamos un sencillo ejemplo:

```
#!/bin/sh
# Esto es un comentario
echo "Los usuarios conectados son:"
who
```

La primera línea indica al *shell* que va a ejecutar un *script* y para ello va a usar un intérprete definido (en este caso, *sh*, que es el que se utiliza por defecto si esta primera línea se omite). Podemos ejecutar este fichero de varias formas:

- (a) Creando un *shell* no interactivo y redireccionando su entrada:

```
rosita:~$ sh < fichero
```

- (b) Creando un *shell* no interactivo y pasándole el *script* como argumento:

```
rosita:~$ sh fichero
```

- (c) Dando permiso de ejecución al *shellscript* y ejecutándolo:

```
rosita:~$ chmod +x fichero
rosita:~$ ./fichero
```

Podemos crear *scripts* para muchas necesidades básicas que como usuarios vamos a tener. Sin embargo, el estudio en profundidad de la programación bajo el *shell* se escapa del tiempo y los propósitos de este curso. No podemos entrar en la posibilidad de recibir argumentos en un *script*, ni en el uso de las variables en la programación *shellscript*, etc. Simplemente hemos de saber cómo interpretar algún programa sencillo (para ello basta conocer mínimamente las órdenes de Unix), y saber cómo crear los nuestros propios (también conociendo los mandatos del sistema).

7.6. Organización de directorios

La estructura de directorios que proporciona Unix nos va a permitir gestionar nuestros propios archivos. Esta estructura consiste en una serie jerárquica de niveles de directorios; todos parten del directorio `'/'`, llamado raíz, y se ramifican con archivos o subdirectorios.

La jerarquía de directorios y archivos lógica puede implantarse en diferentes medios físicos (discos), ya que Unix es un sistema independiente de dispositivo (no como MS-DOS, por ejemplo). De esta forma, el directorio `/home/toni/programacio/` puede estar en un disco y el directorio `/home/toni/security/` puede estar en otro diferente, sin que nosotros notemos el cambio de disco al pasar de un directorio a otro.

Los nombres de los directorios del sistema Linux (no los que nosotros vayamos a crear como usuarios) tienen un nombre definido que da una idea de los archivos planos contenidos en el directorio. Veamos algunos ejemplos:

- `/dev/` (*device*): Dispositivos del sistema (impresoras, discos, módems. . .).
- `/bin/` (*binary*): Ejecutables básicos para los usuarios.
- `/sbin/` (*super binary*): Ejecutables básicos para el superusuario (*root*).
- `/lib/` (*libraries*): Librerías del sistema.
- `/home/` : Directorio del que van a 'colgar' los directorios asignados a los diferentes usuarios. Por ejemplo, `/home/toni/` es el directorio al que el usuario *toni* entra por defecto, y `/home/mbenet/` es al que *mbenet* entra por defecto.
- `/etc/` : Directorio para almacenar programas del sistema (configuración, servicios. . .) que no tienen una localización específica en otro directorio.
- `/proc/` : Dispositivos hardware del sistema (memoria, procesador . . .). Podremos ver características del sistema haciendo un `cat` sobre algunos de ellos (la mayoría).
- `/usr/` (*user*): Directorio del que 'cuelgan' subdirectorios con aplicaciones para usuarios; por ejemplo, en `/usr/bin/` hay archivos ejecutables, en `/usr/include` ficheros de cabecera para programación en C, etc.
- `/var/` (varios): Directorio del que parten subdirectorios con contenidos diversos, desde archivos de administración del sistema hasta correo de usuarios.

En clase veremos con algo más de detalle el contenido de cada uno de estos directorios.

Hemos comentado que cada usuario del sistema tiene un directorio de trabajo al cuál irá por defecto cada vez que conecte a la máquina. Este directorio de conoce como *$HOME* (*'$'* indica que es una variable de entorno), y está especificado en el archivo `/etc/passwd`. Si ejecutamos `cd` sin ningún argumento, iremos a nuestro *$HOME* sin importar nuestra localización actual. Así mismo, para ir al directorio *$HOME* de un usuario (si los permisos asignados lo permiten), bastará con teclear `cd ~<usuario>`:

```
rosita:/usr/include/linux$ cd ~toni
rosita:~$
```

Por último, veamos lo que representa la variable de entorno *$PATH*. El *path* o ruta de un archivo es el camino que se ha de seguir a lo largo de la jerarquía de directorios hasta llegar al fichero; por tanto, cada *path* va a representar siempre a un archivo y sólo a uno (aunque Unix permite que un mismo archivo tenga varios nombres en la jerarquía de directorios, como vimos al estudiar la orden `ln`, el estudio en profundidad de estos conceptos escapa al contenido del curso).

Quizás nos hemos hecho la siguiente pregunta: *cuando nosotros ejecutamos un mandato, sin indicar el path de la orden, ¿cómo sabe el sistema donde buscar el fichero para ejecutarlo?*. Así, si ejecutamos la orden `passwd`, ¿cómo sabe el sistema que nos estamos refiriendo a `/bin/passwd` y no a `/etc/passwd`? Pues bien, el sistema lo sabe simplemente porque nosotros se lo hemos dicho, le hemos indicado dónde ha de buscar los ficheros para ejecutarlos. Esto se lo hemos dicho en forma de una variable llamada *$PATH*. Siempre que indiquemos un mandato al sistema, él buscará un ejecutable con el mismo nombre en cada uno de los directorios del *$PATH*, hasta encontrar el que buscábamos y ejecutarlo. Esta búsqueda se realiza por orden, por lo que si nuestro *$PATH* es como el que sigue,

```
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11/bin:/usr/andrew/bin:
/usr/openwin/bin:/usr/games:/usr/local/bin/varios:
/usr/local/bin/xwpe-1.4.2:.
```

y hay un archivo ejecutable llamado `tester` en el directorio actual, y otro en el directorio `/usr/local/bin/`, y tecleamos simplemente `tester`, se ejecutará `/usr/local/bin/tester`, y no `./tester`. Esto es una importante medida de seguridad para los usuarios.

7.7. Planos de trabajo

En cualquier sistema Unix vamos a tener dos planos de trabajo diferenciados entre sí: el plano principal (*foreground*) y el segundo plano (*background*).

Los trabajos o procesos que tengamos trabajando en segundo plano lo van a hacer generalmente con menos prioridad que los del plano principal; además, si tenemos que interactuar con la tarea del segundo plano (a través del teclado, por ejemplo), va a ser necesario 'subir' el trabajo al plano principal; por tanto, el *background* nos va a ser de utilidad sólo si el proceso no tiene acción con el usuario o si esta acción es cada bastante tiempo (como a la hora de acceder a un servidor WWW cuando hay bastante tráfico en la red). En segundo plano podremos tener tantos trabajos como queramos (o como el sistema nos permita), pero en *foreground* sólo podremos tener una tarea por sesión.

Existen varias formas de dejar trabajos en uno u otro plano y de moverlos entre ellos; vamos a ver aquí las más comunes:

Si ejecutamos una orden, sin indicarle nada especial al sistema, pasará a ejecutarse en *foreground*. Si durante la ejecución queremos moverlo al segundo plano, lo habremos de detener (*Ctrl-Z*), y luego pasarlo con la orden *bg %<número_trabajo>*. El argumento de la instrucción `bg` es opcional; si no lo indicamos, bajará a *background* el último trabajo lanzado. Si tenemos varios trabajos en segundo plano, y queremos seleccionar uno diferente del último para subirlo al plano pincipal, veremos sus números con la orden `jobs`, así como si tenemos diferentes trabajos parados y queremos dejar uno de ellos en *background* o *foreground*.

En el mismo *prompt* del sistema, podemos dejar directamente un trabajo en *background* tecleando al final de la orden (opciones incluidas) el símbolo `'&'`; se nos indicará el identificador de proceso (PID), y el trabajo comenzará a funcionar en segundo plano. Luego podemos subirlo a *foreground* de nuevo con la orden `fg`, pero sin necesidad de suspenderlo. Vamos a ver un ejemplo que nos aclarará las dudas:

```
rosita:~/myprog$ ls
```

```
mysocket.h   add          fsi.c        pingkrc.c    serversock   trans.c
add.c        hora         serversock.c ttylock      addus.c      hora.c
ttylock.c    asker        log          asker.c      log.c        quest
dimoni.h     log2.c       quest.c      sysinfo.c
```

`rosita:~/myprog$ cc -o server serversock.c`
Lanzamos un trabajo en foreground.

`rosita:~/myprog$ server &`
[1] 106
Ahora hemos lanzado un trabajo, nuestro número [1], en background, con el símbolo &; se nos indica el PID: 106.

`rosita:~/myprog$ jobs`
[1]+ Running server &
Vemos nuestra lista de trabajos; sólo tenemos uno.

`rosita:~/myprog$ fg`
server
[1]+ Stopped server
Subimos el trabajo a foreground con fg (podríamos haberlo hecho con fg %1) y lo paramos con Ctrl-Z.

`rosita:~/myprog$ bg`
[1]+ server &
Lo dejamos en background de nuevo, con bg (o bg %1).

`rosita:~/myprog$ cc -o server2 serversock.c &`
[2] 107
Ejecutamos una tarea directamente en background; su número es el 2, y su PID el 107.

`rosita:~/myprog$ fg%2`
cc -o server2 serversock.c
Dejamos el segundo trabajo en el plano principal.

`rosita:~/myprog$ jobs`
[1]+ Running server &
Una vez que ha finalizado el trabajo anterior, volvemos a tener únicamente una tarea ejecutándose.

Con este ejemplo ha de quedar ya completamente claro el concepto de los planos de Unix, y como manejarlos. Ahondaremos más en este tema en clase.

7.8. Entrada y salida

En nuestro sistema vamos a tener definidas una entrada estándar (*stdin*, teclado), una salida estándar (*stdout*, pantalla), y una salida de errores (*stderr*, también la pantalla). Sin embargo, es posible que en algún momento nos interese enviar el resultado de una orden a un fichero, o los errores al compilar un trabajo a otro; entonces habremos de redireccionar las salidas y/o entradas, utilizando para ello los símbolos '<' (menor que) y '>' (mayor que).

Para redireccionar *stdout*, habremos simplemente de ejecutar el mandato deseado, y al final de la línea de orden, especificar > `fichero` para enviar la salida a un archivo; si este archivo ya existe, con '>' se sobreescribirá; si no deseamos sobreescribir el fichero, utilizaremos '>>', en lugar de '>'.

Si lo que deseamos es redireccionar la entrada, para que en lugar de leer órdenes del teclado el sistema los lea desde un archivo, utilizaremos el símbolo '<' al final del mandato y sus opciones en la línea del intérprete de órdenes.

Para redireccionar la salida de errores, lo haremos análogamente a *stdout*, pero habremos de indicar un 2 (el identificador de *stderr*) antes de los símbolos '>' o '>>'.

Si lo que deseamos es enviar ambas salidas a un mismo fichero, una forma fácil de hacerlo es utilizando el símbolo `'>&'` (no hay que confundir `'&'` con el indicador de lanzamiento de un proceso en segundo plano).

Veamos unos ejemplos que aclaren todos estos conceptos:

```
rosita:~# ls
mail seguridad Unix documentos
```

Redireccionamos la salida y sobreescribimos 'fichero':

```
rosita:~# ls >fichero
```

El contenido de 'fichero' es simplemente el resultado de ejecutar la orden 'ls', análogo a lo obtenido antes en pantalla.

```
rosita:~# cat fichero
mail seguridad Unix documentos
```

Volvemos a redireccionar, añadiendo a 'fichero':

```
rosita:~# ls >>fichero
```

Ahora 'fichero' tiene su contenido anterior y el nuevo:

```
rosita:~# cat fichero
mail seguridad Unix documentos
mail seguridad Unix documentos
fichero
```

Redireccionamos ahora la entrada:

```
rosita:~# ftp <entrada &
[1] 129
```

Ahora redireccionamos stderr, la salida de errores:

```
rosita:~# cc -Wall -o test test.c 2>errores
```

El contenido de 'errores' es el resultado de compilar el fichero test.c:

```
rosita:~# cat errores
Line 3: Undefined symbol Int.
Line 5: Warning: fopen() redefined.
rosita:~#
```

Si ahora queremos redirigir ambas salidas al mismo fichero, lo conseguimos de la siguiente forma:

```
rosita:~# cc -Wall -o test test.c >& ambas
```

Además de esta capacidad para redirigir la salida o la entrada a archivos (o a dispositivos como discos flexibles o cintas magnéticas, ya que recordemos que en Unix estos periféricos son archivos), podemos realizar sobre la salida de una instrucción lo que se denomina un filtrado: en lugar de

obtener directamente el resultado de una orden por pantalla, podemos aplicarle antes otra orden y mostrar así el resultado final. Recursivamente, este nuevo resultado también podríamos pasarlo por otro programa antes de mostrarlo en pantalla, o redirigirlo a un archivo exactamente tal y como hemos hecho antes.

Para realizar este filtrado en Unix disponemos de lo que se llama *pipes* o tuberías, utilizando el símbolo '|'. La salida de la orden a la izquierda de '|', compuesta por un conjunto de instrucciones o por una sóla, es filtrada por la orden a la derecha del símbolo antes de mostrarse en pantalla.

Veamos un ejemplo, recordando que el mandato `sort` sin argumentos ordenaba alfabéticamente, y la orden `cat` volcaba el contenido de un fichero a la salida estándar; en primer lugar, volcamos el contenido del fichero '`telefo`' a pantalla:

```
rosita:~$ cat telefo
Antonio 3423243
Luis    977895
Juan    3242143
Amalio 332210
Joaquin 234234
Pepa 336544
Ana 91-555234
rosita:~$
```

Ahora, en lugar de mostrar el resultado en pantalla directamente, lo ordenamos antes utilizando la orden `sort`:

```
rosita:~$ cat telefo|sort
Amalio 332210
Ana 91-555234
Antonio 3423243
Joaquin 234234
Juan    3242143
Luis    977895
Pepa 336544
rosita:~$
```

Finalmente, en lugar de mostrar el resultado ordenado en pantalla, guardamos el resultado en un fichero:

```
rosita:~$ cat telefo|sort >telefo.ordenado
rosita:~$
```

En el uso correcto de tuberías encontramos una de las fuentes de potencia del sistema operativo: en una sola línea de órdenes podemos conseguir cosas que desde otros operativos requerirían un gran trabajo o simplemente sería imposible realizar. Utilizar Unix correctamente reside muchas veces en saber cuándo y cómo resolver problemas mediante la combinación de programas utilizando el filtrado.

8. SEGURIDAD BÁSICA DEL USUARIO

8.1. Sistemas de contraseñas

Para verificar la identidad de cada usuario de una máquina Unix se utilizan *passwords*, contraseñas establecidas por el propio usuario y que sólo él debe conocer para evitar que otra persona pueda entrar haciéndose pasar por él, ya que todo lo que el intruso hiciera estaría bajo la responsabilidad del usuario atacado. Para no caer en los errores típicos de muchos libros al tratar el tema de la seguridad del sistema, no llamaremos a los intrusos ni *hackers* ni *crackers*, sino simplemente piratas o intrusos.

Una de las actividades predilectas de muchos piratas novatos es conseguir el archivo de contraseñas del sistema, normalmente `/etc/passwd`, en el que se guardan las claves cifradas de todos los usuarios. El criptosistema usado por Unix es irreversible, esto es, no podemos 'descifrar' las claves. Sin embargo, existen muchos programas capaces de comparar palabras de un diccionario o combinaciones de éstas con todas las contraseñas de los usuarios, avisando si una comparación es correcta. Por tanto, la seguridad del sistema depende en buena medida de las claves utilizadas por sus usuarios.

El archivo de contraseñas normal de un Unix ha de tener permiso de lectura para todo el mundo, para que al intentar acceder a la máquina, el programa *login* pueda comparar la clave introducida con la almacenada en el fichero. Sin embargo, y debido al enorme agujero de seguridad que esto representa, se han diseñado alternativas a este sistema que no permitan a los usuarios leer directamente el archivo de *passwords*. La más conocida de estas alternativas es la conocida como **shadow password**.

En sistemas con *shadow*, el archivo `/etc/passwd` no contiene ninguna clave, sino un símbolo (normalmente '*' o '+') en el campo correspondiente. Las claves reales, cifradas, se guardan en `/etc/shadow`, un archivo que sólo el superusuario puede leer. De esta forma, aunque un intruso consiga el archivo */etc/passwd* no podrá intentar un ataque contra las contraseñas cifradas para romper la seguridad de la máquina. Una entrada típica en el archivo `/etc/passwd` de un sistema *shadow* puede ser

```
toni:*:34:101:Toni Villalon:/home/toni:/bin/bash
```

Vemos que en el segundo campo, el reservado a la clave encriptada, no aparece nada que el intruso pueda violar.

Otra medida, bastante menos utilizada que la anterior, consiste en la técnica denominada *aging password*, que se puede utilizar independientemente o en combinación con el *shadow password*.

El *aging password* por sí mismo aún permite la lectura de las claves encriptadas en `/etc/passwd`. Sin embargo, las claves van a tener un periodo de vida (en el caso más extremo una contraseña servirá simplemente para una única sesión, habiendo de cambiarla en la siguiente). Gracias a esto, se consigue que un intruso que haya capturado una clave no pueda acceder de forma indeterminada al sistema.

Cada cierto tiempo, el sistema va a ordenar a cada usuario cambiar su contraseña. Después de cambiarla, existirá un periodo en el que no la podrá volver a cambiar, para evitar que vuelva a poner la vieja inmediatamente después de cambiarla. Después de este periodo, se permite un cambio de forma voluntaria durante otro intervalo de tiempo, al finalizar el cual se obliga al usuario de nuevo a realizar un cambio. Gráficamente, lo podemos ver como:

(a): Cambio de clave obligado por el sistema.
(b): Periodo sin posibilidad de cambiar la contraseña.
(c): Periodo de cambio voluntario.
(d): Periodo de cambio obligatorio.
(e): Si al finalizar el periodo anterior el usuario no ha cambiado su clave, se le deniega el acceso al sistema (por norma general).

Hemos hablado antes de las herramientas utilizadas por intrusos para conseguir claves del sistema. Para evitar que puedan conseguir nuestro *password*, es conveniente utilizar una contraseña aceptable. No debemos utilizar palabras que puedan estar en un diccionario de cualquier idioma. Debemos utilizar una combinación de símbolos, mayúsculas, minúsculas y números para dificultar así el trabajo de un posible atacante. Aunque las herramientas usadas por los piratas son cada día más potentes, elegir una buena contraseña puede dar al sistema un mínimo de seguridad y fiabilidad.

A nadie le gusta que alguien fisgue en sus archivos, aunque no llegue a destruir nada; sería como si una persona leyera nuestras cartas y no las quemara, sino que las volviera a guardar como si nada hubiera pasado: no hemos de apoyar nunca a alguien dedicado a acceder a sistemas de forma no autorizada, aún cuando nos parezca un genio. Generalmente, nadie con unos mínimos conocimientos se dedica a estas actividades (aunque, como en todo, siempre hay excepciones...).

8.2. Archivos *setuidados* y *setgidados*

Un archivo *setuidado* (o *setgidado*) no es más que un fichero con el bit *setuid* (respectivamente, *setgid*) activo. ¿Qué significa esto? Simplemente que quien lo ejecute tendrá los privilegios del dueño o del grupo del dueño, dependiendo de si es *setuidado* o *setgidado*, durante un cierto intervalo de tiempo.

Este tipo de archivos es una gran fuente de problemas para la integridad del sistema, pero son completamente necesarios. Por ejemplo, imaginemos que `/bin/passwd` no tuviera el bit *setuid* activo; no nos permitiría cambiar nuestra clave de acceso, al no poder escribir en `/etc/passwd`, por lo que cada vez que deseáramos cambiar la contraseña habríamos de recurrir al administrador. Esto sería imposible para un sistema Unix medio, con aproximadamente quinientos o mil usuarios.

Casi todos los medios de acceso no autorizado a una máquina se basan en algún fallo de la programación de estos archivos. No es que los programadores de Unix sean incompetentes, sino que es simplemente imposible no tener algún pequeño error en miles de líneas de código.

Para aumentar la fiabilidad del sistema, es conveniente reducir al mínimo indispensable el número de ficheros *setuidados* o *setgidados*. En el servidor seguramente no tendremos, por ejemplo, la posibilidad de cambiar nuestro *shell*, o nuestro campo de información de usuario en `/etc/passwd`, ya que el bit *setuid* de los ficheros `chsh` y `chfn`, respectivamente, ha sido reseteado.

Para crear un archivo *setuidado* hemos de cambiar el modo del fichero a un 4XXX (*XXX* indica el modo normal; hemos de anticipar el 4), y para crear uno *setgidado* hemos de establecer un modo 2XXX al fichero; veamos unos ejemplos:

```
rosita:~# chmod 4755 test
```

```
rosita:~# chmod 2745 test2
rosita:~# ls -al
-rwsr-xr-x   1 root    root         155 Nov  3 05:46  test
-rwxr-Sr-x   1 root    root         349 Oct  3 05:14  test2
rosita:~#
```

Vemos que un archivo *setuidado* tendrá una *'s'* en el campo de ejecución del propietario, y un *setgidado* una *'s'* en el campo de ejecución del grupo al que pertenece el creador del fichero. Si lo que encontramos en cualquiera de estos campos es una *'S'*, este dato nos está indicando que, aunque el archivo ha sido *setuidado* o *setgidado*, no se ha activado el permiso de ejecución correspondiente, por lo que no vamos a poder ejecutar el fichero.

Como ejemplo de la compartición de privilegios, si alguien tuviera un *shell setuidado* con nuestro UID, habría conseguido un intérprete de órdenes con todos nuestros privilegios, lo que le permitiría borrar nuestros archivos, leer nuestro correo, etc.

8.3. Privilegios de usuario

Cada usuario de la máquina Unix tiene una serie de privilegios sobre unos determinados archivos y directorios. Existe un usuario, llamado *root*, que tiene acceso a toda la información del sistema, ya que actúa como administrador de la máquina.

Nosotros, como usuarios normales, vamos a tener capacidad para escribir en nuestro directorio *$HOME*, cambiar su modo, borrar sus archivos y subdirectorios, etc. Sin embargo, nuestro acceso a ciertos directorios o archivos del sistema va a ser limitado (lectura, ejecución, o a veces ni tan siquiera eso...). Podremos acceder al *$HOME* de otros usuarios, si éstos lo han permitido, y leer alguno de sus archivos, pero generalmente no vamos a poder modificar su contenido.

Tampoco vamos a poder leer ciertos archivos que están dedicados a registrar actividades del sistema, a modo de auditoría de seguridad, como pueden ser `/var/adm/syslog` o `/var/adm/messages`, o cualquier otro fichero de uso interno de la administración del sistema (como `/etc/sudoers`).

Existe, sin embargo, un directorio aparte de nuestro *$HOME* en el que sí vamos a poder escribir y guardar nuestros ficheros: se trata de `/tmp`, dedicado a uso temporal por parte de todos los usuarios del sistema, en el que podremos guardar durante un tiempo información que nos sea de interés.
Aunque todos pueden escribir en `/tmp`, sólo nosotros y el administrador vamos a poder borrar los archivos que hayamos creado; esto se consigue gracias al *sticky bit*, o bit de permanencia, del directorio (si lo listamos con `ls -l`, veremos que su modo es `drwxrwxrwt`; ésta última *'t'* es el *sticky bit*).

La función del *sticky bit* en un archivo va a ser ligeramente diferente: va a indicarle al sistema que el fichero es frecuentemente ejecutado y por ello es conveniente que esté la mayor parte del tiempo en memoria principal. Para evitar problemas en el sistema, aunque cualquier usuario puede conseguir que el bit aparezca en sus ficheros, éste sólo va a ser realmente efectivo cuando ha sido el administrador el que lo ha activado.

8.4. Cifrado de datos

El cifrado de la información es la técnica más utilizada para garantizar la privacidad del los datos. La Criptología es la ciencia ocupada del estudio de los cifrados en todas sus variantes, desde los ataques hasta el desarrollo de nuevos métodos de encriptación (criptosistemas).

La encriptación de un fichero hace imposible a un intruso la lectura de los datos si no posee una clave para desencriptar el archivo; por tanto, garantiza la privacidad de una información. Su uso es

hoy en día muy importante, no sólo ya en sistemas militares o de inteligencia, sino en el comercio a través de redes o en los sistemas de dinero electrónico.

En Unix, se utiliza el cifrado de datos principalmente en la protección de las contraseñas (ya hablamos de ello en el primer apartado de esta sección). En algún momento nos puede interesar cifrar un archivo para que nadie más que el poseedor de una clave pueda leer su contenido. Para ello, tenemos a nuestra disposición varios sistemas, todos ellos demasiado complejos para ser tratados en este curso. En muchos servidores está instalado PGP, un sistema de firma electrónica que garantiza la privacidad e integridad de la información. En otras redes de computadores más complejas es posible encontrar sistemas de verificación como *Kerberos*, en los cuales todo el tráfico de la red viaja cifrado, lo que hace inútil algunas actividades comunes entre los intrusos, como el uso de *sniffers*, programas que capturan los paquetes de información que transitan por un ordenador. Como ejemplo del cifrado de las transmisiones, tenemos disponible en nuestra máquina accesos por SSH (*Secure Shell*) o SSL-Telnet (*Secure Socket Layer Telnet*).

8.5. Bloqueo de terminales

No es conveniente dejar nuestra conexión abierta si abandonamos, aunque sea momentáneamente, el ordenador o la terminal desde el que hemos conectado. Cualquiera podría, por ejemplo, copiar en unos segundos todos nuestros ficheros, borrarlos, o conseguir un *shell setuidado* con nuestro UID, que como comentamos en el apartado 2, le daría todos nuestros privilegios permanentemente.

Por tanto, la mejor solución si nos vemos obligados a abandonar el ordenador desde el que estamos trabajando es salir del servidor. Si por cualquier motivo no nos interesa salir del sistema, existen a nuestra disposición unos programas que bloquean el teclado hasta que introduzcamos una clave determinada. Para el modo texto en el que vamos a trabajar, en nuestro caso tenemos *termlock*, que nos va a preguntar una clave para posteriormente desbloquear la terminal. De este modo, vamos a evitar que alguien aproveche nuestra ausencia para modificar de cualquier forma nuestra sesión de trabajo. Trabajando en consola, la terminal principal del sistema, o con sistemas gráficos como *X-Window*, tenemos bloqueadores de terminal mucho más complejos, aunque el objetivo principal siempre va a ser el mismo: evitar dejar una conexión abierta sin un usuario responsable a la otra parte del teclado.

Referencias

[1] Blanco *Linux: Instalación, administración y uso del sistema*
Ra-Ma, 1996

[2] Hahn *Internet. Manual de referencia*
McGraw-Hill, 1994

[3] Kernighan,Pike *The Unix programming environment*
Prentice Hall, 1984

[4] Poole, Poole *Using Unix by example*
Addison Wesley, 1986

[5] Ribagorda, Calvo, Gallardo *Seguridad en Unix: Sistemas abiertos e Internet*
Paraninfo, 1996

[6] Rosen, Rosinski, Host *Best Unix tips ever*
McGraw-Hill, 1994

[7] Salus *A quarter century of Unix*
Addison Wesley, 1994

[8] Silberschatz, Peterson, Galvin *Sistemas Operativos. Conceptos fundamentales*
Addison Wesley, 1994

[9] Siyan *Internet Firewalls and Network Security*
Prentice Hall, 1995

[10] Stevens *Unix Network Programming*
Prentice Hall, 1990

[11] Tanenbaum *Sistemas Operativos: Diseño e implementación*
Prentice Hall, 1987

[12] Welsh, Kauffman *Running Linux*
O´Reilly and Associates, 1995