Universidad de Las Palmas de Gran Canaria

Sistemas Operativos

Guía de operación en entorno UNIX

Escuela Universitaria de Informática

Por: José Miguel Santos Espino
Departamento de Informática y Sistemas

© José Miguel Santos Espino

Impreso en el Servicio de Reprografía del Departamento de Informática y Sistemas. Campus Universitario de Tafira, Las Palmas de Gran Canaria, febrero de 1997.

Prohibida su reproducción.

ISBN. 84-8098-032-X
Depósito legal GC 52-1997

Tipografía

La tipografía empleada en este manual sigue en general estas convenciones:

negrita	concepto importante que aparece por vez primera
courier	líneas de órdenes, ejemplos de uso, algoritmos
itálica	parámetros en líneas de órdenes
cursiva	concepto resaltado o bien texto en inglés

Terminología

Como principio general, se han traducido todos los términos ingleses al español, salvo en casos que complicarían la comprensión, debido al uso extensivo del término inglés. Un ejemplo es la palabra *shell* para referirse a los procesadores de órdenes.

En cualquier caso, los términos ingleses se presentan junto con los castellanos, habida cuenta de que la documentación de los sistemas UNIX (y de todo lo informático) se halla escrita mayormente en la lengua de Shakespeare.

Índice

1. Conceptos generales sobre UNIX

En esta primera sección se dará una visión general del sistema operativo UNIX, sentando ciertas bases útiles para comprender el sistema en su conjunto. Se detallan más bien los aspectos que tienen una relación fuerte con lo que será la operación y la administración del sistema. Así pues, las características más centradas en la programación de sistemas, o el desarrollo en general, no serán contempladas más que superficialmente.

En adelante se presupone que el lector tiene conocimientos sobre informática a nivel de usuario y operador. Es aconsejable un mínimo de conocimientos de programación. También ha de conocer los fundamentos de un sistema operativo: sus funciones y sus componentes más habituales. Más en concreto se asume que el lector tiene una experiencia previa como usuario de otros entornos de operación, como MS-DOS o Windows sobre PCs.

2. Panorámica histórica

El sistema operativo UNIX nace en 1969 fruto de la iniciativa personal de Ken Thomson, un programador de la división Bell Laboratories, de AT&T. Bell Labs había abandonado hacía unos años el proyecto MULTICS, que pretendía crear un sistema operativo capaz de dar suministro doméstico de potencia de computación, de forma análoga a un servicio de teléfonos. Ken Thomson retoma muchas ideas de este sistema para confeccionar un pequeño sistema operativo con el propósito de hacerle el trabajo más agradable y, según la leyenda, poder implementar un juego de marcianitos de cosecha propia.

Este nuevo y modesto sistema operativo, que alguien llamó UNIX, se extendió rápidamente entre los trabajadores de Bell Labs. La empresa AT&T impulsa el nuevo sistema, cediendo recursos al equipo de Ken Thomson para perfeccionar el UNIX. Entre 1972 y 1974 se reescribe el UNIX en un lenguaje de alto nivel llamado C, concebido por Dennis Ritchie para ese fin. El UNIX ha sido el primer sistema operativo escrito en un lenguaje de alto nivel, rompiendo la hasta entonces imposición de trabajar en ensamblador.

A partir de entonces, la historia del UNIX y del lenguaje C corren parejas.

A mediados de los setenta (1976) AT&T distribuye gratuitamente el UNIX por las universidades americanas. Entrega los fuentes en C del sistema, con lo que los investigadores de USA pueden experimentar con él, descubrir y corregir fallas, crear nuevas utilidades, etc. Al estar escrito en C, se puede adaptar a cualquier máquina, lo que acelera su difusión.

En esos años el UNIX se robustece con las enmiendas de los investigadores y se enriquece con un sinnúmero de utilidades creadas por equipos en universidades o por programadores particulares. De esta forma consensuada, el UNIX va añadiendo características no impuestas por una empresa matriz, sino amoldándose a las necesidades de sus propios usuarios.

En 1980 la Universidad de Berkeley lanza el UNIX BSD, que incluye por primera vez memoria virtual y soporte para Internet (TCP/IP). Esta versión coexiste con la versión 7 *oficial*. Por estas fechas, se estima que el 80% de los centros de USA utiliza UNIX.

En 1982 se funda Sun Microsystems, que con estaciones de trabajo operando con UNIX terminará por inundar el mercado de la gama media de computadores, haciendo tambalear a DEC, que hasta entonces dominaba ese segmento con sus sistemas VAX-VMS.

Finalmente, en 1983 AT&T decide lanzar su primera versión comercial de UNIX, el Sistema V (SV). Ya los fuentes no son públicos: la etapa universitaria ha concluido definitivamente. El Sistema V difiere sensiblemente de la versión BSD, apareciendo el riesgo de pérdida de transportabilidad de los programas.

Desde mediados de los ochenta, comienza una etapa para el UNIX de consolidación y estandarización, en la que se define la norma POSIX, de IEEE, para garantizar la transportabilidad de las aplicaciones UNIX, reconciliando los dos principales dialectos, el SV y el BSD. Otra norma similar es la XDG.

Ya los finales de los ochenta y principios de los noventa suponen la ubicuidad del UNIX, el cual consigue que empresas como Digital e IBM, cuya política había sido de mantener sus propios sistemas operativos, pasen a ofrecer sus propias versiones de UNIX: ULTRIX y AIX, respectivamente. El UNIX queda

en la actualidad convertido en un estándar oficioso de sistema operativo a escala mundial.

El sistema operativo con el que trabaja el mayor número de máquinas hoy día es el Windows de Microsoft. A pesar del dominio de Microsoft en la informática personal, existen versiones de UNIX para PCs que corren en una buena porción del parque informático. Y en cualquier caso, la influencia del UNIX se deja ver en los sistemas operativos de Microsoft.

2.1 Conclusiones

El UNIX partió de la iniciativa personal de un programador, para hacer la vida más fácil a los programadores.

Dispone de muchas y variadas herramientas para compilación, edición, manipulación de textos y control de procesos.

Por eso el UNIX es un ENTORNO DE DESARROLLO de aplicaciones o un ENTORNO DE PROGRAMACIÓN.

Como desventaja, su interfaz de usuario es pésima. Sólo recientemente han surgido entornos gráficos amigables, como el OPEN LOOK de Solaris o el VUE del HP-UX.

Otra desventaja es que su esquema de protección y seguridad es demasiado simple, insuficiente en ciertos casos. Los sistemas comerciales suelen proporcionar utilidades de su propia cosecha para cubrir esta carencia.

El resultado actual del UNIX es fruto de una vasta combinación de esfuerzos en universidades, empresas y otras instituciones; muchas de sus utilidades han sido aprobadas por consenso de los usuarios.

- El sistema se ha refinado y resulta muy fiable
  (o al menos previsible)
- Han prevalecido las utilidades más eficientes y aceptadas
- El sistema es poco modular, sobre todo lo referente a aplicaciones del sistema
- Se deja sentir poca cohesión en el conjunto de utilidades

El UNIX está escrito en C (lenguaje de alto nivel) y se conocen prácticamente todos los algoritmos y estructuras que lo componen.

- Es muy transportable de una arquitectura a otra
- Existen muy pocas "áreas oscuras" o comportamientos no documentados

3. Estructura y funciones del UNIX

3.1 Rasgos generales del entorno UNIX

Características ventajosas:

- Sistema operativo universal, utilizable en máquinas grandes y pequeñas
- Transportable al estar escrito en C
- Sistema abierto: cualquier empresa puede desarrollar su versión de UNIX
- Concebido por y para programadores: entorno de desarrollo

Características desventajosas:

- Pobre sistema de administración
- Sistema para *gurús*: escasa atención a la interfaz de usuario o ayuda
- Falta de uniformidad en los programas del sistema

3.2 La cultura UNIX

El UNIX trasciende el ámbito de los sistemas operativos, ya que ha impuesto una forma de considerar los sistemas informáticos y ha condicionado el desarrollo de utilidades y lenguajes. Por tanto es lícito hablar de una *cultura UNIX*.

En torno a la órbita del UNIX encontramos:

- El lenguaje C y el C++
- La profusión de las redes: la Internet y el WWW
- La prestigiosa casa de software lgratuito GNU
- La interfaz gráfica XWindows
- Estándares de gráficos 2D/3D (PHIGS)
- Sistemas operativos distribuidos (Mach, Chorus)

3.3 Características del sistema

El sistema operativo UNIX se caracteriza por ofrecer, entre otros, estos servicios o funciones:

- Sistema de ficheros jerárquico
- Manejo uniforme de la entrada/salida, empotrada en el sistema de ficheros
- Multiprogramación interactiva por *tiempo compartido*
- Entorno multiusuario
- Memoria virtual paginada

- Soporte de comunicaciones a través de red

El UNIX estándar no ofrece este tipo de servicios:

- Procesamiento en tiempo real
- Tolerancia a fallos
- Procesamiento distribuido

No obstante, existen extensiones de UNIX que proveen en mayor o menor medida estos servicios.

3.4 Estructura del UNIX

La estructura de un sistema es la forma en que está constituido: cuáles son sus componentes y de qué manera funcionan para conseguir sus propósitos. Sobre el UNIX podemos decir que se compone de los siguientes módulos:

- Núcleo o kernel
- Bibliotecas de funciones
- Procesadores de órdenes o shells
- Programas del sistema

Estos módulos se encuentran entre el *hardware* de la máquina y el usuario (como en cualquier sistema operativo), de una forma que puede representarse con un gráfico de niveles concéntricos cada vez más alejados de la máquina y más cercanos al usuario:

3.5 Usuarios en UNIX

El UNIX es un sistema **multiusuario**: se distinguen múltiples usuarios con distintas capacidades de uso del sistema.

Cada usuario tiene asociada una **cuenta**.

Cada usuario tiene ciertos **permisos** y restricciones sobre las operaciones en el sistema.

Antes de entrar a trabajar en el sistema, hay que autentificarse como usuario, escribiendo una cuenta y una contraseña secreta.

Existe un usuario de máximo privilegio: el **superusuario.**

Cada objeto en UNIX (proceso o fichero) tiene un usuario **propietario**.

Cada usuario pertenece a un **grupo** de usuarios.

3.6 Clases de usuarios

Pueden distinguirse tres clases de usuarios, según el tipo de tareas que desempeñe:

El administrador	Administración del sistema, mantenimiento, instalación de nuevos productos
El operador	Manejo cotidiano de las aplicaciones
El programador	Desarrollo de nuevas aplicaciones

Este texto se va a centrar principalmente en el rol de operador.

3.7 Objetos de UNIX

El UNIX distingue dos tipos de entidades:

- Procesos
- Ficheros

Un **proceso** es un programa en ejecución.

Un **fichero** es un depósito de datos que tiene un nombre.

Todo objeto en UNIX tiene un usuario **propietario**.

Sobre cada objeto del sistema -especialmente los ficheros- se establecen unos **permisos** de uso, que otorgan o revocan privilegios de uso sobre los posibles usuarios del sistema.

4. El sistema de ficheros

UNIX proporciona al usuario un sistema de ficheros, que sirve para almacenar todo tipo de datos de forma permanente: documentos, bases de datos, programas fuentes y programas ejecutables, etc. Los ficheros se organizan de forma jerárquica en directorios.

4.1 Conceptos sobre ficheros

Un fichero en UNIX es una secuencia de bytes que posee un nombre por el cual es accesible.

Al contrario de otros sistemas, un fichero carece de estructuración interna: el sistema operativo UNIX no permite definir registros o tipos de datos en sus ficheros.

Se distinguen varias clases especiales de ficheros (directorios, dispositivos, etc.), según se explica más adelante.

4.1.1 Nombres de ficheros

Todo fichero en UNIX tiene un nombre, que es una secuencia de caracteres. Algunas características de los nombres de ficheros en UNIX son:

- distingue entre mayúsculas y minúsculas
- un nombre de fichero en UNIX es simplemente una tira de caracteres. No se distingue nombre y extensión
- los nombres de ficheros pueden tener hasta 255 caracteres (o hasta 14 en ciertos sistemas)
- algunos caracteres, aunque en teoría posibles, no se deberían emplear: *, ?,/, la coma, el punto y coma, el espacio, porque sirven para otros fines

los caracteres no ASCII, es decir, letras acentuadas, la eñe, etc. se pueden emplear si el sistema los reconoce, aunque no es recomendable.

Ejemplos de nombres válidos	**Ejemplos de nombres no válidos**
pepe.c	*asteriscos*
pepe.C	tiene espacios
.pepe.c	?no debería?
pepe.c.antiguo	coma,peligrosa
nombre_de_fichero.MUY_LARGO	

4.2 Organización del sistema de ficheros. Directorios

Los ficheros se agrupan en un sistema de ficheros jerárquico en forma de árbol.

Al comienzo de ese árbol se encuentra el **directorio raíz** (*root*).

Un **directorio** es un conjunto de entradas que pueden ser ficheros u otros directorios.

Los directorios que penden de un directorio D se denominan **subdirectorios** de D.
A su vez, se dice que D es el **directorio padre** de esos subdirectorios.

Todo directorio contiene dos entradas de particular interés.

- **.** (punto) apunta a ese mimo directorio.
- **..** (punto, punto) apunta al directorio padre.

Hay operaciones para crear (**mkdir**) y borrar (**rmdir**) directorios.

4.2.1 Nombres de directorios

Los directorios son una clase especial de ficheros. Por tanto, pueden tener los mismos nombres que se han descrito para los ficheros regulares.

4.2.2 Rutas de acceso a ficheros

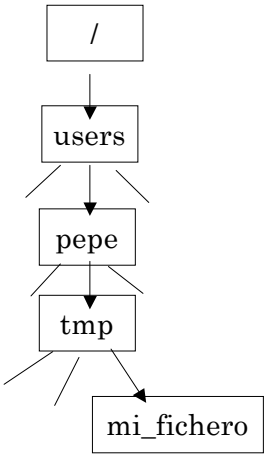
Para acceder a un fichero, se ha de detallar su **ruta** (*path*), que es el camino para llegar al mismo partiendo del directorio raíz. Cada "salto" en el camino va separado por una barra '/'.

Ruta absoluta: completa desde el directorio raíz. Comienza por /
Ejemplo: **/users/pepe/tmp/mi_fichero**

4.2.3 Directorio actual de trabajo. Rutas relativas

El **directorio actual de trabajo** (CWD) especifica un punto cualquiera dentro del sistema de ficheros. Sirve para especificar rutas más cortas, a partir de ese punto, que se llaman *rutas relativas*.

Ruta relativa: ruta desde el directorio actual de trabajo.
No comienza por /
Ejemplo: si el CWD es **/users/pepe**, el ejemplo anterior puede ser **tmp/mi_fichero**



Existe una operación para mover el directorio actual de trabajo a cualquier lugar del sistema de ficheros. Se llama **cd**.

Ejemplo:

cd /users/pepe	va a /users/pepe
cd tmp	va a /users/pepe/tmp
cd ..	va a /users/pepe (padre de /users/pepe/tmp)

4.2.4 Directorio local

Cada usuario tiene asignado un **directorio local** (*home directory*), que habitualmente es de su propiedad y donde presumiblemente estará la mayoría de sus ficheros.

Cuando un usuario inicia una sesión, su directorio actual de trabajo se mueve a su directorio local.

4.3 Integración de múltiples sistemas de ficheros

Un sistema informático típico posee varios dispositivos de almacenamiento secundario. Por ejemplo, varios discos duros, una disquetera, un lector de CD-ROM, etc. Cada dispositivo puede contener su propio sistema de ficheros. Es más, un disco duro puede contener *particiones*, cada una con su árbol de directorios.

En UNIX todos estos sistemas de ficheros están integrados en el único árbol de directorios. Sólo existe un directorio raíz. Cada sistema de ficheros tiene la apariencia de un subdirectorio.

Durante el arranque del sistema, o en otro momento posterior, los sistemas de ficheros se **montan** como subdirectorios.

Algunos sistemas de ficheros no son de carácter permanente, como es el caso de un CD-ROM: sólo hay información cuando hay insertado un CD en el lector, y la información desaparece cuando se extrae.

Al insertar un disco, hay que *montarlo* como subdirectorio en el sistema de ficheros. Al extraer el disco, hay que *desmontarlo* para evitar posteriores problemas.

4.4 Clases de ficheros. La entrada/salida en el sistema de ficheros

Todos los dispositivos de entrada/salida son accesibles como si fueran ficheros (normalmente instalados en el directorio **/dev**). En UNIX, operar con un dispositivo se convierte en leer o escribir en un fichero.

Suponiendo que existe un fichero **/dev/lp0** que está asociado con una impresora, la orden

cp fichero /dev/lp0

copia el contenido de *fichero* en la impresora.

Así, el UNIX proporciona una interfaz uniforme de uso de la entrada/salida, incorporándola en el concepto de fichero.

4.4.1 Clases de ficheros

El sistema operativo distingue los ficheros *regulares* (ficheros de datos convencionales) de los fichero *especiales*, como pueden ser los dispositivos de entrada/salida. De esta forma una operación de lectura o escritura sobre un fichero se puede convertir en una operación de entrada/salida.

Los directorios se consideran una clase especial de ficheros.

Entre otros, los tipos de ficheros que distingue UNIX son:

- Ficheros regulares
- Directorios
- Dispositivos (de bloques o de caracteres)
- Tuberías (*pipes*)
- Enlaces simbólicos

4.5 Directorios estándares

La organización del sistema de ficheros en UNIX cumple ciertas normas. Los ficheros ejecutables suelen hallarse en determinados directorios, así como los dispositivos, ficheros temporales y otros.

A continuación se muestran algunos directorios que cumplen misiones específicas en UNIX.

/	directorio raíz
/bin, /usr/bin	programas del sistema
/etc, /usr/etc	programas y ficheros de administración
/dev	dispositivos de E/S
/mnt	sistemas de ficheros montados
/lib, /usr/lib	bibliotecas de funciones para compiladores
/usr	programas adicionales no estándares
/tmp, /usr/tmp	archivos temporales
/lost-found	archivos perdidos y recuperados tras un daño
/var	directorios con información del sistema que tiende a crecer (auditorías, correo, ficheros para imprimir, etc.)
/home, /users	directorios locales de usuarios

© *José Miguel Santos Espino*

5. Protección y seguridad en ficheros

Todo fichero tiene un **usuario propietario**, y un **grupo propietario**.
Todo fichero tiene definidos unos permisos de uso, según estas tres clases:

R	permiso de lectura (incluye acceso a su contenido y copia)
W	permiso de escritura (incluye modificación, borrado y apéndice)
X	permiso de ejecución (el fichero es ejecutable)

Los permisos se declaran por separado para estas tres clases de usuarios:

Para el propietario (*user*)
Para el grupo propietario (*group*)
Para el resto de los usuarios (*others*)

Los permisos, pues, se definen mediante una tira de nueve bits:

R W X	R W X	R W X
USER	GROUP	OTHERS

En un directorio, el permiso de ejecución se interpreta como **permiso de búsqueda**. (*search*). Si un directorio tiene permiso de ejecución para un usuario, éste puede trabajar con rutas que contengan tal directorio. Si por el contrario, el directorio no le da permiso de ejecución, el usuario no podrá nombrar ficheros que requieran atravesar ese directorio.

El superusuario puede prescindir de los permisos de cualquier fichero
El propietario de un fichero puede alterar los permisos de éste (programa **chmod**).

6. Operación básica en UNIX

6.1 Sesiones de trabajo

El sistema operativo UNIX es multiusuario, es decir, reconoce distintas clases de usuarios con diferentes privilegios y restricciones de uso del sistema. Cada usuario Unix está representado con una **cuenta** de usuario, que le faculta para entrar a trabajar en el sistema de forma controlada.

Cada fichero y cada proceso en el sistema tienen un usuario propietario, que puede efectuar cualquier operación sobre ellos. Sobre los ficheros se pueden especificar permisos de uso para todos los usuarios.

Cuando se pretende entrar a trabajar en un sistema Unix, hay que *autentificarse* como usuario válido en el sistema, aportando el nombre de la cuenta y una contraseña (*password*, en inglés).

Además, cuando se termina de trabajar en un entorno UNIX, hay que declararlo expresamente; de lo contrario el sistema creerá que el usuario aún se encuentra en la terminal donde entró (y alguien podría suplantarle).

Se llama sesión de trabajo, o simplemente **sesión**, al proceso de entrar en el sistema, trabajar en él y salir. Las sesiones de trabajo exigen al menos un protocolo de entrada y otro de salida.

La entrada en el sistema se denomina en inglés *login*.

La salida del sistema se denomina en inglés *logout*.

Durante la sesión, existe un proceso que interpreta las órdenes escritas del usuario. Se le llama **procesador de órdenes** o *shell* en inglés, y básicamente lo que hace es recoger información desde el teclado, interpretarla y ejecutar los programas correspondientes.

El *shell* actúa como interfaz entre el usuario humano y el núcleo del sistema operativo.

6.1.1 Abreviaturas

Existen algunos nombres especiales para ciertos directorios de interés.

/	es el directorio raíz
.	es el directorio de trabajo
..	es el directorio padre del de trabajo

6.2 Sobre el uso del teclado

Para la edición de líneas y ciertos controles el UNIX reconoce una serie de combinaciones de teclas, usualmente de la forma tecla CONTROL + tecla alfa-numérica. En la documentación de UNIX, una combinación de CONTROL + tecla se escribe ^*tecla*.

Las combinaciones reconocidas por el UNIX dependen de la configuración de la terminal y de las características del procesador de órdenes que estén ejecutando. Sí existen unas costumbres generales, como estas:

^C	interrumpe la ejecución de un programa
^S	congela la salida de datos por pantalla
^Q	reanuda la salida de datos por pantalla

Los cursores, tecla de borrado, etc., pueden funcionar o no. Experimente y actúe en consecuencia.

En esta guía nos referiremos a la tecla de entrar línea como INTRO. En UNIX, el carácter generado por esa tecla se conoce como carácter de "nueva línea" (*newline*).

7. Los procesadores de órdenes

Cuando ingresamos en un sistema UNIX, se ejecuta un programa denominado *procesador de órdenes* que se dedica a leer órdenes tecleadas por nosotros y a ejecutarlas una vez interpretadas. El vocablo inglés para referirse a un procesador de órdenes es *shell* o caparazón, por la idea de que "recubre" al núcleo del sistema operativo. A partir de ahora se emplearán indistintamente el término *shell* o "procesador de órdenes".

En general, las órdenes suponen la ejecución de un programa ejecutable. Las órdenes son líneas de caracteres que habitualmente se introducen mediante el teclado de la terminal y se denominan **comandos** u **órdenes**. El término *comando* es una traducción impropia de la voz inglesa *command*, aunque ha penetrado en la literatura informática en español.

Las siguientes páginas son en buena medida una guía de referencia de uso de los *shells*, en la que se describen con cierta profundidad muchas de sus herramientas.

7.1 Tipos de procesadores de órdenes

En UNIX existen varios procesadores de órdenes estándares, los cuales difieren en la sintaxis de las órdenes, pero prácticamente con la misma potencia. Los tres procesadores estándares son:

sh	Bourne shell
csh	C shell
ksh	Korn shell

El **sh** ha sido el *shell* original de UNIX. El **ksh** es una extensión del **sh**, casi coincidente con el *shell* que define la norma POSIX. También existen otros *shells* no estándares, pero igualmente extendidos, como el **bash**.

En este texto nos referiremos sobre todo al "Korn shell", **ksh**.

7.1.1 Sintaxis de las órdenes

Cualquier entorno de operación hace suyo un determinado "estilo de trabajo", que hace más simple imaginar cómo trabajar con un programa sin disponer de información previa. En definitiva, se hace el sistemas más previsible. En este sentido, las órdenes en UNIX tienen una sintaxis uniforme.

Las órdenes se lanzan al procesador de órdenes tecleándolas y terminando con INTRO; este grupo de caracteres para interpretar se denomina **línea de órdenes** o **línea de comandos**.

Una orden elemental en UNIX siempre está formada por el nombre de la orden (que casi siempre es un fichero ejecutable), a veces seguida por uno o más *argumentos* o *parámetros*.

El UNIX distingue mayúsculas; no es lo mismo **cat** que **Cat**.

En UNIX casi todas las órdenes y programas del sistema están en minúsculas.

En la línea de órdenes pueden aparecer varias órdenes elementales separadas por punto y coma ";", que se ejecutarán secuencialmente (una detrás de otra).

Una orden UNIX casi siempre adopta esta sintaxis:

orden ***[opciones] [argumentos] [ficheros]***

Los corchetes indican que las tres clases de parámetros son opcionales.

7.2 Opciones (*switches*)

Las opciones suelen presentan la sintaxis ***-letra*** o ***-letra1letra2...*** Es decir, precedidas del carácter guión. Indican características o modos de trabajo que el programa ejecutado activará o desactivará, sin importar por lo general el orden. Estos ejemplos suelen ser equivalentes:

```
programa -a -b -c
programa -b -ca
programa -abc
```

7.3 Argumentos y ficheros

Los argumentos pasados a un orden son datos con los que operar (leer, crear, transformar, etc.) Un argumento puede ser un fichero; en general, si coexisten argumentos que no sean ficheros (p.ej. un nombre de usuario) con otros que sí lo sean, los ficheros aparecen al final del orden.

Ejemplos:

```
ls -l /bin /usr/pepe
chown pepe /usr/pepe/fichero.c
```

8. Una primera sesión

En estas líneas se plantea, a modo de ejercicio interactivo, una primera sesión bajo UNIX, donde se introducirán conceptos y órdenes muy comunes y elementales.

8.1 Entrada en el sistema (*login*)

Para entrar en el sistema, es imprescindible autentificarse como un usuario válido del mismo. Para ello hay que aportar un nombre de cuenta y una contraseña (*password*) en forma de clave alfanumérica.

Cuando una terminal o estación está inactiva, presenta un mensaje similar a

login:

Teclee entonces el nombre de su cuenta y pulse INTRO. A continuación el sistema le pedirá su contraseña con

passwd:

Teclee la contraseña y pulse INTRO. Observe que los caracteres de la clave no aparecen por pantalla, para evitar aviesas miradas de otros usuarios.

El sistema contrasta el nombre de la cuenta y la clave con la información del fichero de texto **/etc/passwd**, donde está la lista de todos los usuarios, sus claves y otros datos relacionados.

Si el sistema encuentra una cuenta y una clave correctas, se inicia la sesión, ejecutándose un procesador de órdenes (*shell*).

Si se ha cometido algún error, o no se es un usuario autorizado, se volverá a solicitar la entrada con **login:** . En ciertos sistemas existe un máximo número de intentos o un tiempo límite para autentificarse, pasados los cuales se notifica al sistema y se reanuda el proceso de entrada.

En caso de hallarse en un entorno gráfico, el inicio de la sesión hará ejecutarse un entorno de ventanas (por ejemplo, OPEN LOOK o VUE) cuya interfaz permitirá dar instrucciones a través tanto del teclado como del ratón.

8.2 Primeras órdenes

Ingrese en el sistema del modo descrito anteriormente. Si ha entrado satisfactoriamente, el sistema operativo lanza un *shell* que espera a que usted le encomiende tareas, habitualmente en forma de expresiones escritas desde el teclado.

El *shell* anuncia su disposición a recibir órdenes mediante un texto (*prompt*) que puede ser algo así como un **$** o una línea parecida a:

sopa[1]$

Habitualmente, cuando se entra como *root*, el *prompt* que aparece es un carácter almohadilla:

```
#
```

Tenga precaución en ese caso, ya que el superusuario tiene capacidad de destruir toda la información del sistema.

Comience por ejecutar una orden sencilla. Entre la palabra **who**. En la terminal aparecerán unas líneas parecidas a esto:

Se muestran los distintos usuarios que están activos en el sistema, con detalles de cuánto tiempo llevan trabajando, desde qué terminal, etc.

Lo que ha hecho el *shell* es ejecutar un programa llamado **who**, que se encuentra como fichero (ejecutable) en algún punto del sistema de ficheros. La inmensa mayoría de las órdenes en UNIX son ficheros ejecutables que se hallan en determinados directorios del sistema, como **/bin** o **/usr/bin**.

Compruebe cuál es su directorio actual de trabajo tecleando **pwd**:

```
$ pwd
/home/pepe
$
```

Esta orden escribe en pantalla la ruta absoluta del directorio actual de trabajo (CWD). Como hasta ahora no ha alterado el CWD, ha de coincidir con su directorio local (*home directory*), en este caso **/home/pepe**.

8.3 Primeras operaciones con ficheros

Realizará ahora operaciones elementales con ficheros. Por ejemplo, el programa **ls** lista el contenido del directorio actual. Teclee **ls**. Lo más probable es que no aparezca nada, ya que cuando se entra por primera vez en el sistema, no existen ficheros en el directorio local (en realidad sí existen unos cuantos, pero se verán más adelante).

Ahora pruebe con

```
ls /etc
```

Esta orden lista los ficheros del directorio **/etc** (note que el directorio viene especificado con una ruta absoluta).

El programa **ls** admite varias opciones; pruebe a teclear

ls -l /etc/passwd

Le mostrará un listado detallado del fichero en cuestión, con información sobre el propietario del fichero, su longitud en bytes, la fecha de última modificación, etc.:

-rw-r--r-- 1 root root 19401 Jun 9 18:38 passwd

El convenio en general del UNIX es pasar las opciones a los programas con letras antecedidas de un guión, como en este caso.

8.4 Sobre usuarios y protecciones

Observe que el propietario del fichero **/etc/passwd** es un usuario llamado "root", que es el superusuario. Normalmente el resto de usuarios no tienen derecho a modificar o borrar ese fichero; esta información viene expresada por los caracteres -r--r--r-- o bien -rw-r--r-- que aparecen dentro de la salida del programa anterior.

8.5 Navegación por el sistema de ficheros

Pruebe ahora a moverse hacia otro directorio. Teclee **cd /etc**; hará que su CWD pase a ser el **/etc**. Compruébelo tecleando **pwd**. Observe que ahora puede ver el fichero **/etc/passwd** tecleando

ls -l passwd

es decir, una ruta relativa a partir de **/etc**. Experimente con la orden **cd** "navegando" por el sistema de ficheros. Sepa que

cd / le lleva al directorio raíz
cd . le lleva al directorio actual (le deja donde ya estaba)
cd .. le lleva al directorio padre del actual
cd le lleva al directorio local de su cuenta

8.6 Ficheros ocultos

Vuelva a su directorio local. Hecho esto, teclee

ls -a

Ahora sí aparecen líneas por pantalla. Vea que aparecen los ficheros **.** y **..** indicando respectivamente el directorio actual y el padre. Aparte, pueden aparecer algunos

ficheros con nombres tales como **.profile** o **.login**, que sólo se visualizan con la opción **-a** (que significa *all*).

Los ficheros con nombres que comienzan por un punto son utilizados por programas del sistema para distintos propósitos (configuración sobre todo) y no suelen aparecer en los listados de directorios para no molestar.

8.7 Algunas órdenes más

Por probar algo más antes de salir, teclee **date**. Este programa le muestra el día y la hora actuales.

Si en cualquier momento quiere "limpiar" la pantalla, teclee **clear**.

8.8 Salida de la sesión (*logout*)

Por fin, para salir de la sesión puede emplear varios métodos, que funcionarán o no según circunstancias ahora mismo difíciles de explicar. Puede probar tecleando ^D. Esto informará al *shell* de que ya no le suministrará más órdenes y lo obligará a terminar.

Otra forma más elegante de salir es tecleando **exit**.

En algunos sistemas se puede abandonar la sesión con el programa **logout**.

Si funciona la salida de la sesión, volverá a aparecer el mensaje de bienvenida y la palabra

login:

para iniciar una nueva sesión.

IMPORTANTE: NUNCA abandone el sistema sin realizar este proceso de salida. No se limite a apagar la terminal: su sesión de trabajo persistirá cuando se vuelva a encender. Si deja su sesión abierta está a expensas de accesos indebidos por parte de otras personas.

NUNCA concluya su trabajo simplemente apagando la máquina. Esto seguramente destruirá datos aún no almacenados, y puede que incluso corrompa el sistema.

9. Una sesión más avanzada

Esta sección también planteará como ejercicio desarrollar una sesión, en la que se introducirán nuevas órdenes. Las secciones que vienen a continuación explican de una forma más organizada distintos aspectos de la operación del UNIX.

Entre de nuevo en el sistema de la forma habitual.

9.1 Cambio de contraseña

Para cambiar su contraseña, introduzca **passwd**. Se le pedirá la clave antigua, y luego la nueva, que tendrá que reescribir por seguridad.

Algunos sistemas exigen que la clave tenga una longitud mínima, que contenga al menos un carácter no alfabético (para dificultar que otros descubran nuestra contraseña).

IMPORTANTE: NUNCA, NUNCA utilice palabras previsibles en sus contraseña (mismo nombre de la cuenta, número de DNI, nombre del compañero/a sentimental). Cuanto más sencilla sea su contraseña, más probable es que alguien la adivine.

9.2 Visualización de ficheros

Pruebe ahora a visualizar el contenido de uno de esos ficheros. Teclee

cat /etc/passwd

El fichero **/etc/passwd** contiene la información de los usuarios del sistema, con sus nombres, claves cifradas, directorios locales, etc.

Si este fichero, o cualquier otro, es lo bastante largo, la pantalla no habrá sido lo suficientemente larga para ver por completo el contenido, y las primeras líneas habrán desaparecido.

Para ver el fichero poco a poco, puede usarse la combinación de teclas ^S y ^X. Una manera más sensata es teclear

more /etc/passwd

Cada vez que los caracteres llenan la pantalla, la salida de caracteres se interrumpe y espera por que el usuario pulse una tecla. Con INTRO se avanza línea a línea, mientras que la tecla de espacio provoca el avance pantalla por pantalla.

9.3 Copia de ficheros

Pasemos a crear un fichero. En concreto, copiaremos el **/etc/passwd** a nuestro directorio local. Esto se puede conseguir tecleando

cp /etc/passwd passwd

o bien

cp /etc/passwd .

El programa **cp** sirve para copiar ficheros.

En el primer ejemplo, se especifica el fichero para copiar (fichero origen) y a continuación el fichero que va a aparecer como copia (fichero destino).

En el segundo ejemplo, el segundo argumento es un punto, que es el nombre del directorio actual. Lo que realiza **cp** en este caso es una copia del fichero **passwd** que se encuentra en **/etc** hacia el directorio actual, creando un fichero (local) llamado también **passwd**.

Si ahora teclea usted **ls** aparecerá **passwd** como único fichero en su directorio actual. Tecleando **ls -l**, podrá visualizar más información referente al nuevo fichero.

Observe que la información de longitud es idéntica al fichero original, pero hay cambios significativos. Por ejemplo, el propietario y el grupo ya no son **root** y **sys**, respectivamente, sino el nombre del usuario correspondiente a su cuenta, y el grupo al que pertenece tal cuenta. Usted, o mejor dicho su cuenta, es el propietario del fichero copiado.

Otra variación menos importante es la fecha de última modificación.

9.4 El programa cat

Para disponer de algún fichero más, se puede utilizar el programa **cat**. Teclee

cat >pepe

La terminal se quedará "colgada" en espera de que usted entre líneas de texto. Hágalo y para terminar, pulse ^D. Habrá creado un fichero de texto llamado **pepe**. Teclee ahora **ls -l** para visualizar el contenido del directorio actual.

Continúe con este método hasta disponer de unos cuantos ficheros.

9.5 Subdirectorios. Creación y borrado

Acto seguido, va a crear un subdirectorio. Asegúrese de encontrarse en su directorio local. Hecho esto, teclee

mkdir mi_carpeta

lo que creará un directorio llamado **mi_carpeta**. El programa **mkdir** crea directorios; **rmdir** los borra.

9.6 Más sobre las órdenes UNIX

El programa **cp** admite una sintaxis más versátil que la expuesta. Por ejemplo, si existen los ficheros **pepe** y **juan** en el directorio actual, se pueden copiar al subdirectorio **uno** de esta forma:

```
cp pepe juan mi_carpeta
```

Es decir, permite especificar múltiples ficheros como parámetros, al contrario que el MS-DOS, con una sintaxis más rígida.

En UNIX muchas órdenes admiten especificar múltiples argumentos. Por ejemplo, el programa **mv** mueve o renombra ficheros. Si hubiese querido mover, en lugar de copiar los ficheros del ejemplo anterior, habría tecleado

```
mv pepe juan mi_carpeta
```

Ahora pruebe a copiar ficheros en el nuevo subdirectorio; experimente con las rutas absolutas, relativas, etc. y utilice **cd** para moverse de un sitio a otro. Por ejemplo, cree un directorio llamado **copia,** muévase a él y luego copie los ficheros que hay en **mi_carpeta** a **copia.**

9.7 Borrado de ficheros

Para borrar ficheros, utilice el programa **rm**, como se muestra en el ejemplo:

```
$ ls
pepe
juan
directorio
copia
$ rm pepe
$ ls
juan
directorio
copia
```

Trate ahora de borrar el fichero **/etc/passwd**.

```
$ rm /etc/passwd
/etc/passwd: permission denied
$
```

. Lo que habrá ocurrido es que no ha podido borrar el fichero, dado que ni nos pertenece (el propietario es el usuario **root**), ni el fichero tiene permiso de escritura para su cuenta.

Si intenta borrar un fichero de su propiedad, pero para el que no tiene permisos de escritura, es posible que se le muestre un mensaje como

rm: remove fichero?

Si teclea **y**, se procede al borrado.

Para borrar todos los ficheros del directorio **copia**, teclee

rm copia/*

Muchas veces interesa borrar *recursivamente* un directorio, es decir, todos sus ficheros, más los ficheros de los subdirectorios de ese directorio, y así hasta eliminar por completo todo ese subárbol de ficheros. Emplee la opción **-r**:

rm -r *directorio*

IMPORTANTE: en UNIX no existe ninguna utilidad estándar de recuperación de ficheros borrados. Utilice el programa **rm** con prudencia, asuma que lo que borre habrá desaparecido para siempre.

9.8 Ayuda en línea

Si desea obtener más información sobre cualquier orden, teclee

man *orden*

y se visualizará la documentación sobre la ***orden***, procedente de los manuales que se hallan en el disco (*en línea,* que se dice).

9.9 Final

Con este ejemplo, usted se habrá familiarizado con unas cuantas órdenes del UNIX y su entorno de operación. Estas y otras órdenes de interés irán apareciendo en las secciones de referencia que vienen a continuación.

10. Protecciones de ficheros

Como ya se ha dicho, en UNIX todo fichero tiene un propietario, más un grupo de propietarios, que poseen una serie de permisos de uso sobre el mismo.

Veamos una posible salida de **ls -l**:

```
$ ls -l
drwxrwxrwx  2 jomis   users     1024 Jun 14 10:56 .
drwxr-xr-x 10 bin     bin       1024 Jun 14 19:12 ..
-rw-rw-r--  1 jomis   users     9401 Jun  9 18:38 pepe.c
-r-xr-xr-x  1 jomis   users     3174 Jun 14 10:53 hola.out
```

El primer campo es una retahíla de letras y guiones que informa sobre el tipo de fichero y sus permisos.

10.1 Clases de ficheros

La primera letra del listado largo indica el tipo de fichero que se muestra. Aunque el UNIX distingue varias clases de ficheros, para las operaciones cotidianas basta conocer estas dos:

d el fichero es un directorio
- el fichero es un fichero regular ("normal")

Es útil, sin embargo, saber que existen estos otros posibles valores:

c el fichero es un dispositivo de E/S de caracteres
b el fichero es un dispositivo de E/S de bloques
p el fichero es un conducto o *pipe*
l el fichero es un enlace simbólico a otro fichero (un sinónimo)

10.2 Protecciones

Los siguientes 9 caracteres son interpretados como tres tripletes de tres bits cada uno, en el orden RWX (lectura, escritura y ejecución). El primer triplete RWX indica los permisos del propietario; el segundo, los permisos del grupo propietario, y el último los permisos del resto de usuarios.

RWX	**RWX**	**RWX**
usuario	grupo	otros

Los caracteres que aparecen se interpretan como sigue, dependiendo de si se trata de un fichero regular o un directorio:

Ficheros regulares:

r el fichero es legible por el sujeto en cuestión (propietario, grupo, otros)

w el fichero es alterable por ese sujeto

x el fichero es ejecutable

Directorios:

r se puede ver su contenido

w se pueden añadir o borrar ficheros dentro de él

x permiso de búsqueda (*search*): se puede "visitar" o "atravesar"

Si un permiso no está concedido, en su lugar correspondiente aparece un guión '**-**'.

En el listado de arriba, el fichero **pepe.c** es un fichero regular cuyo propietario es el usuario *jomis*, el grupo propietario es *users*, y declara los siguientes permisos. *jomis* puede leer y escribir el fichero; los usuarios que pertenezcan a *users* pueden también leer y escribir, y el resto de usuarios tan sólo pueden leer del fichero.

Los permisos sobre directorios tienen cualidades muy particulares. Si un usuario no tiene permiso de lectura para un directorio, no puede hacer un **ls** sobre él, aunque si sabe los nombres de sus ficheros y tiene permiso de lectura sobre ellos, puede verlos.

Si se retira el permiso de ejecución en un directorio, es como si se levantara un *muro* en el sistema de ficheros. Quienes carezcan del permiso, no podrán ni ver lo que hay bajo el directorio con **cd** o **ls**; ni tampoco usar nombres de ficheros que contengan en su ruta a ese directorio.

El superusuario puede saltarse todos los permisos sobre un fichero o directorio.

10.3 Cambio de permisos: chmod

El programa **chmod** permite al propietario de un fichero, o al superusuario, alterar los permisos del mismo. Su sintaxis es

chmod ***permisos fichero***

El fichero puede ser un fichero regular o un directorio. Existen varias formas de declarar los nuevos *permisos*, que contamos a continuación.

Forma octal (indicada para informáticos)

Se toman los nueve permisos como tres números consecutivos de 3 bits cada uno. Un bit vale 1 si su permiso correspondiente está activado y 0 en caso contrario. Cada número se expresa en decimal, del 0 al 7, y los permisos quedan definidos como un número octal de tres dígitos.

Por ejemplo, los permisos **rw-r--r-x** son el número octal 645.

Forma simbólica

Se utilizan las letras **r,w** y **x** para los permisos. Las letras **u**,**g**,**o** y **a** para especificar el tipo de usuario, y los caracteres **+** y **-** para declarar si se añade o se revoca el permiso.

Si se utiliza un carácter igual **=**, se dejan sólo los permisos indicados en **chmod**, y el resto se eliminan.

u	el usuario propietario
g	el grupo propietario
o	otros usuarios (no propietarios)
a	todos los usuarios

Con unos ejemplos:

chmod go-w fichero

retira (**-**) el permiso de escritura (**w**) al grupo propietario (**g**) y al resto de los usuarios (**o**).

chmod ug=rwx fichero

concede todos los permisos al usuario y al grupo propietario. Los restantes usuarios quedan sin derecho alguno sobre el fichero.

La opción **-R** permite alterar los permisos de un directorio recursivamente. Observe que es una "r" mayúscula.

chmod -R u=rw /users/pepe

cambia los permisos de todo el árbol de directorios a partir de **/users/pepe**; sólo el propietario puede leer y escribir.

10.4 Cambio de propietario

En algunos sistemas se puede ceder la propiedad de sus ficheros a otro usuario, o grupo. Se realiza con los programas **chown** y **chgrp**:

chown ***usuario ficheros***
chgrp ***grupo ficheros***

chown y **chgrp** también admiten una opción **-R** para que el cambio afecte recursivamente descendiendo por los directorios pasados como argumentos. Hay que usar este programa con precaución, dado que una vez cedida la propiedad, no se puede recuperar.

El programa **chown** está restringido al superusuario en varias versiones de UNIX.

11. Los procesadores de órdenes (*shells*). Referencia básica.

En este capítulo se da una guía de referencia sobre características elementales y algo más avanzadas de los procesadores de órdenes. Se estudiarán, entre otras herramientas, las redirecciones, conductos y utilidades interactivas.

Otro aspecto de los *shells*, los **escritos de *shell***, que son una manera rápida y efectiva de construir pequeños programas bajo UNIX, se estudiarán en detalle en el capítulo 16.

11.1 Los procesos y el *shell*

11.1.1 Procesos en UNIX

Cualquier orden o programa que se esté ejecutando en un momento dado es un proceso. Un proceso es un programa en ejecución.

Puede haber varios procesos simultáneamente sobre el mismo programa (por ejemplo, tres usuarios utilizando el mismo editor).

Todo proceso posee un **identificador de proceso (PID)**.

Todo proceso proviene de un proceso que lo lanzó. A éste se le llama **proceso padre**. De esta forma se constituye una genealogía de procesos padres y descendientes. Un proceso puede tener varios hijos, pero un hijo sólo un padre.

11.1.2 Cómo ejecuta los procesos el *shell*

El *shell* que atiende nuestras órdenes es un proceso más en el sistema.

Cuando se solicita la ejecución de una orden, como **ls**, el procesador de órdenes (*shell*) lanza un proceso como hijo suyo. Así, el proceso hijo y el *shell* se ejecutan concurrentemente. Lo que ocurre es que el *shell* espera por el hijo lanzado hasta que este último termine. Por eso, si tecleamos **ls**, no aparece el símbolo (*prompt*) del *shell* hasta que finaliza el listado de ficheros.

El *shell* sabe qué proceso ha de lanzar en base a la orden que hayamos especificado. Busca en ciertos directorios un programa que coincida con el nombre de la orden. Si, por ejemplo, se teclea **ls**, el *shell* ejecuta el programa **/usr/bin/ls**. Más adelante se explica cómo sabe el *shell* donde buscar los programas.

11.1.3 Constatación: el programa `ps`

El programa **ps** muestra los procesos que están en ejecución. Si no se le pasan opciones, sólo nos lista los procesos asociados a nuestra sesión de trabajo actual:

```
$ ps
PID TT STAT  TIME COMMAND
600 p0 S    0:00 ksh
616 p0 R    0:00 ps
```

(Los campos del listado pueden variar según la versión de UNIX).

Vea que hay (al menos) dos procesos: uno es el **ksh**, que es el procesador de órdenes. El otro es el propio **ps**, que precisamente se estaba ejecutando mientras se escribía ese resultado.

La columna PID muestra los identificadores de los procesos. La columna TT especifica la terminal adscrita a los mismos; y STAT indica el estado en el que se encuentra cada proceso. La S quiere decir que el **ksh** está durmiendo (*sleeping*), y la R que **ps** está ejecutándose (*running*).

El formato del listado de **ps** puede variar según la versión de UNIX de que se trate.

El programa **ps** es útil para verificar el estado de los procesos de segundo plano (ver siguiente apartado) o si se quiere acabar con procesos "colgados".

11.1.4 Órdenes internas

Hay ciertas órdenes que no conducen al *shell* a lanzar un proceso hijo, sino que son interpretadas y ejecutadas por el propio *shell*. Se las llama **órdenes internas**, que simplemente modifican el estado interno del procesador de órdenes.

Una de las órdenes internas más utilizadas es **cd**.

En MS-DOS muchas órdenes del *shell* son internas (dir, copy, rename...) Casi siempre sus equivalentes en UNIX son programas ejecutables ajenos al *shell*.

11.2 Procesos en segundo plano (*background*)

Habitualmente, cuando se ejecuta una orden con el *shell*, se lanza un proceso interactivo y además no se devuelve el control al *shell* hasta que el proceso no haya concluido. En ciertos casos es deseable, y hasta necesario, lanzar el proceso y que se mantenga en ejecución haciendo sus cosas mientras el usuario prosigue interactuando con el *shell*.

Un proceso que está ejecutándose al margen de la interacción con el *shell* se dice que está en **segundo plano** (*background*).

Los procesos interactivos se llaman de **primer plano** (*foreground*).

Para lanzar un proceso en segundo plano se utiliza el símbolo "ampersand" **&**.

Ejemplo:

```
$ de.fondo&
[1] 398
$
```

El programa **de.fondo** se lanza y se queda ejecutando en segundo plano. Los números que se ven a continuación son el número de tarea y el PID del proceso.

Tras lanzar el proceso, vuelve a aparecer el símbolo del *shell* para seguir introduciendo órdenes.

La opción de dejar procesos en segundo plano es muy práctica cuando se tenga que ejecutar programas que vayan a tardar mucho en completarse y que puedan trabajar de forma no interactiva (sin esperar por entradas tecleadas del usuario).

Ejemplos: formateo de disquetes, copias de seguridad, impresión de documentos, etc.

Con los procesos hay que tener, no obstante, las siguientes precauciones:

- Cuantos más procesos concurran en el sistema, más lento puede ir éste
- La salida a pantalla de los procesos en segundo plano se mezclará con la salida habitual de nuestros programas. Por eso se hace conveniente redirigir la salida estándar de un proceso que se va a lanzar en segundo plano.

Cuando un proceso que estaba en segundo plano finaliza, se envía un mensaje indicativo a la terminal, parecido a éste:

[1] Done programa

11.3 Sustitución de nombres de ficheros: comodines

Para acelerar la escritura de órdenes que afectan a múltiples ficheros, o a veces para descubrir ficheros desconocidos de los que sólo se conoce una parte del nombre, el UNIX permite especificar argumentos empleando "plantillas" o patrones que responden a múltiples nombres.

Cada palabra de la línea de órdenes se procesa como una plantilla para sustituirla por los nombres de ficheros que se ajusten a ella. Las plantillas utilizan ciertos caracteres o construcciones especiales; sin entrar en profundidad, mencionaremos las más importantes.

Dentro de una expresión regular, el carácter asterisco, *****, admite un número indefinido de caracteres cualesquiera. El cierre de interrogación **?** admite cualquier carácter (uno solo).

A estos dos caracteres se les llama **comodines** o *wildcards*.

Un conjunto de caracteres entre corchetes "**[...]**" admite cualquier carácter individual perteneciente al conjunto.

La mejor forma de entender la sustitución de nombres de ficheros es mediante un ejemplo. Supongamos que en el directorio de trabajo existen los ficheros **pepe.c**, **pepe.h**, **pipo.c** y **pipa.x**. Entonces, si introducimos estas órdenes:

ls pepe.*	listará **pepe.c** y **pepe.h**
ls p?p*c	listará **pepe.c** y **pipo.c**
ls p?p?[xc]	listará **pepe.c**, **pipo.c** y **pipa.h**

La práctica totalidad de las órdenes UNIX reconoce estas expresiones regulares. En realidad es el procesador de órdenes el que las procesa previamente a la ejecución de la orden. El *shell*, después de leer una línea de órdenes, procesa los caracteres comodín para crear la orden definitiva. Es entonces cuando se ejecuta el programa.

La sustitución puede servir para abreviar órdenes tediosas de escribir. Si en nuestro directorio actual tenemos estos tres ficheros:

pepe.c
hola
fichero.de.nombre.muy.largo

Para editar el último fichero, bastaría con escribir:

vi fich*

esta orden editará **fichero.de.nombre.muy.largo**, dado que se trata del único que se ajusta a la plantilla **fich***

11.4 Flujos estándares. Redirección de entrada y salida

Un proceso toma y escribe datos desde y hacia el exterior. El *shell*, por ejemplo, lee caracteres del teclado e imprime caracteres en la pantalla. En UNIX, los procesos se comunican con el exterior a través de **flujos** (*streams*).

Conceptualmente, un flujo es una ristra de *bytes* que se puede ir leyendo o sobre la que se puede escribir caracteres. Un flujo puede ser un fichero ordinario, o estar asociado a un dispositivo. Cuando se lee del teclado es porque previamente se ha abierto como flujo de caracteres del que leer. Un proceso, cuando muestra algo por pantalla, está escribiendo caracteres a un flujo de salida.

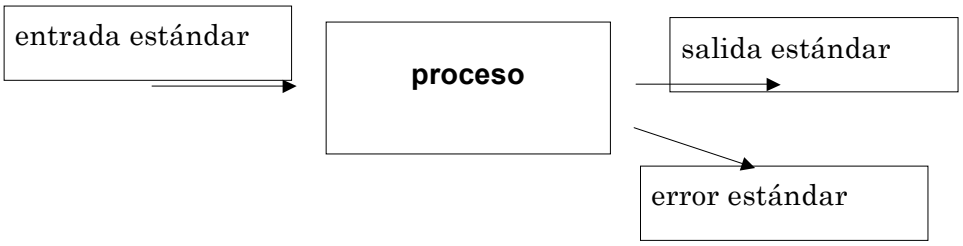
11.4.1 Entrada, salida y error estándares

Los procesos lanzados por un procesador de órdenes (un *shell*) utilizan dos flujos de particular interés: la **entrada estándar** y la **salida estándar**. La entrada estándar es utilizada para recoger información, y la salida estándar para enviar información al exterior.

La entrada estándar suele ser el teclado. La salida estándar suele ser la pantalla.

Por ejemplo, cuando se ejecuta **ls**, este programa nos muestra los ficheros del directorio actual escribiendo los caracteres necesarios sobre la salida estándar (que suele ser la pantalla).

Los flujos estándares pueden redirigirse a otros ficheros, como se verá dentro de unas líneas.



Existe también el llamado **error estándar**, flujo donde se vierten los mensajes de error. Habitualmente coincide con la salida estándar, pero se considera un flujo diferente.

11.4.2 Redirección de entrada y salida

Los flujos estándares pueden ser redirigidos a cualquier fichero del sistema, de forma que un programa no escriba sus resultados en la pantalla, sino sobre un archivo cualquiera. Con este fin se utilizan los signos **<** y **>**, con este significado:

programa <*entrada*	se redirige la entrada estándar desde el fichero ***entrada***
programa >*salida*	se redirige la salida estándar al fichero ***salida***

Con un ejemplo:

```
$ date
Lun 6 Nov 22:44:05 GMT
$ date >fecha
$ cat fecha
Lun 6 Nov 22:44:05 GMT
$
```

Se ha redirigido la salida del programa **date** a un fichero llamado **fecha**.

El fichero de redirección de salida se crea como nuevo; si ya existía, la información precedente se pierde. En algunos sistemas se instala la opción de prohibir la redireción si el fichero ya estaba creado.

Otro ejemplo de redirección de salida: la orden **ls /usr/bin >listado** crea un fichero **listado** con el contenido del directorio **/usr/bin**.

Un ejemplo ilustrativo, tanto de lo que son los flujos estándares como de su redirección, es el programa **cat**. Cuando se llama a **cat** sin parámetros, se comporta como un programa "tonto" que simplemente toma caracteres de la entrada estándar y los escribe en la salida estándar. Por ello, si simplemente se ejecuta **cat**, se duplica por pantalla todo lo que se teclea. Eso hasta que se pulse ^D, que es la indicación de que la entrada estándar "ha terminado":

```
$ cat
hola                    (lo escribe el usuario)
hola
esto es una línea       (lo escribe el usuario)
esto es una línea
^D                      (control+D, final de entrada estándar)
$
```

La combinación Control+D viene a significar “fin de fichero” o, más propiamente, “fin de flujo”, indicando que no hay más caracteres en la entrada estándar.

Si se ejecuta **cat** redirigiendo su entrada y salida estándares, como en

```
cat <pepe >juan
```

la entrada estándar es el fichero **pepe**, y la salida estándar el fichero **juan**. El resultado es que el contenido de **pepe** se copia en **juan**. No es difícil intuir lo que ocurre al redirigir sólo uno de los dos flujos estándares:

cat <pepe	muestra por pantalla el contenido de **pepe**
cat >pepe	recoge caracteres del teclado y los envía a **pepe**

Otro ejemplo más jugoso:

```
cat f1 f2 f3 >todos.juntos
```

Lista los tres ficheros **f1**, **f2** y **f3** en la salida estándar, que es ahora el fichero **todos.juntos**. El resultado es que este último fichero resulta ser la concatenación de aquellos tres. Las redirecciones son herramientas versátiles para resolver muchas operaciones en la labor cotidiana sobre UNIX.

Los signos de redirección se pueden ubicar en cualquier punto de la línea de órdenes. Así, son equivalentes todas estas órdenes:

```
wc -l <datos >resul
wc -l >resul <datos
wc <datos -l >resul
<datos >resul wc -l
```

La primera forma es la más empleada, más acorde con el *estilo Unix*.

Las redirecciones no se consideran parámetros en la línea de órdenes.

11.4.3 Apéndice de ficheros

La combinacion **>>*fichero*** en una línea de órdenes funciona como una redirección de salida, pero con una diferencia: si el fichero de salida ya existía, los datos generados por el programa hacia su salida estándar se escriben al final del fichero.

Ejemplo:

```
$ cat fichero
Este es un fichero con una línea
$ cat >>fichero
No, son dos líneas
^D
$ cat fichero
Este es un fichero con una línea
No, son dos líneas
```

11.4.4 Redirección de errores

El error estándar también se puede redirigir, aunque de manera distinta según el *shell* de que se trate. En el **sh** y el **ksh**, se emplea el signo **2>*fichero***, como en este ejemplo:

```
$ rm /etc/passwd 2>error
$ cat error
cp: /etc/passwd: permission denied
$
```

También se acepta **2>>fichero** para añadir a un fichero información proveniente del error estándar.

En el **csh** no se puede redirigir el error aisladamente; no obstante, se pueden redirigir simultáneamente la salida y el error estándares con **>&*fichero***.

11.5 Conductos o tuberías (*pipelines*)

Después de conocer los flujos estándares y la redirección, se está en situación de presentar una forma de combinar órdenes en UNIX muy característica de este sistema y que le da una potencia expresiva enorme. Son los llamados **conductos**, **tuberías** o *pipelines*.

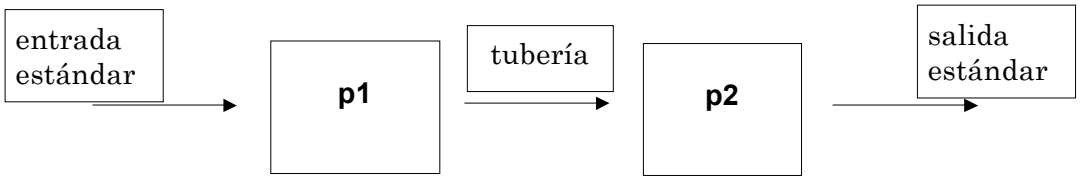
A la vista del gráfico del apartado anterior, se puede considerar un proceso lanzado por

un *shell* como una "tubería" por la que fluyen unos datos de entrada y de salida:

Un conducto de dos órdenes ***p1*** y ***p2*** se expresa en UNIX de esta manera:

p1* | *p2

Si se teclea esto, el *shell* lanzará concurrentemente los procesos **p** y **q**, con la particularidad de que la salida estándar de ***p1*** servirá como entrada estándar a ***p2***, como intenta expresar este esquema:



La conexión entre ***p1*** y ***p2*** es lo que se conoce como tubería (*pipe* en inglés).

Un ejemplo muy frecuente de uso de conductos o *pipelines* es utilizar el programa **more** como extremo final de la tubería. El programa **more**, en ausencia de parámetros, lee líneas de la entrada estándar y las copia a la salida estándar, como el **cat**, pero haciendo una pausa cada vez que se llene la pantalla. Si se escribe

```
ls /bin | more
```

el programa **more** irá recogiendo el producto de la salida estándar de **ls /bin**, que es un listado de ficheros, y lo irá mostrando por la pantalla con pausas.

```
$ ls /usr/bin | more
adb
ar
awk
...
cc
cdf
clear
--- More ---
```

Otro ejemplo puede ser

```
ls | wc -l
```

El segundo miembro del conducto es el progrma **wc**, que cuenta las líneas leídas desde su entrada estándar: en este caso es el resultado de **ls**. Lo que se verá por pantalla será el número de ficheros que tiene el directorio actual:

```
$ ls
pepe.c
bienvenido.txt
juan
$ ls | wc -l
3
$
```

El resultado del ejemplo anterior podría haberse obtenido con estas dos órdenes:

```
$ ls >fichero.temporal
$ wc -l <fichero.temporal
```

En realidad, los conductos se implementan creando un fichero temporal especial que sirve de depósito temporal de los caracteres producidos por el primer miembro del conducto. El ejemplo anterior no es del todo equivalente, porque en este último caso el programa **ls** tiene que terminar para que **wc -l** pueda empezar a hacer algo. En un conducto UNIX, ambos programas se lanzan en paralelo y tan pronto como **ls** dé alguna salida, **wc** la recogerá y empezará a trabajar.

Los conductos pueden ser de tantos procesos como se quiera:

```
cat /etc/passwd | sort | more
```

Este ejemplo lista las cuentas de los usuarios del sistema ordenadas alfabéticamente, con pausa tras cada pantallazo. La ordenación la realiza el programa **sort**.

```
ls -F | grep / | wc -l
```

Esta combinación de órdenes escribe el número de subdirectorios que cuelgan del directorio actual. La opción **-F** en **ls** añade en el listado un carácter **/** a los directorios. El programa **grep** escribe en la salida estándar sólo las líneas que contengan el carácter **/**. Finalmente, **wc** se encarga de contar las líneas que devuelve **grep**.

No se deberían combinar los redireccionamientos de entrada o salida en medio de un conducto. Sí se permite redirigir la entrada estándar en el primer miembro, y la salida estándar en el último.

11.5.1 Filtros

Existen muchísimos programas en UNIX que, como **grep** o **sort**, están preparados para leer de su entrada estándar, procesarla de alguna forma y escribir los resultados por su salida estándar. A este tipo de programas se les suele llamar **filtros**, en el sentido de que procesan la información que les llega y la devuelven "filtrada".

sort "filtra" la entrada estándar ordenando sus líneas. **grep** "filtra" la entrada estándar seleccionando sólo las líneas que respondan a un patrón. **cat** es un filtro "nulo": no hace nada con la información que le llega.

Los ejemplos anteriores demuestran el uso de los filtros. El UNIX está plagado de esta clase de programas, los cuales, usados con ingenio, permiten dar órdenes complicadas al sistema sin necesidad de recurrir a lenguajes de programación.

En la guía de referencia de órdenes se describen algunos de los programas para filtros con más utilidad. La experiencia y muchas visitas a los manuales son las que irán haciendo descubrir la versatilidad de los conductos en UNIX.

11.6 Parámetros del *shell*. Entorno del *shell*

Los **parámetros del *shell*** son un conjunto de símbolos alfanuméricos, al estilo de variables, a los que se les puede dar valores. Se emplean para guardar información útil para el *shell* o para el propio usuario, con la misma filosofía que en un lenguaje de programación.

Algunos de los parámetros los emplea el *shell* para saber cuál es el directorio de trabajo, el directorio local, qué tipo de terminal se está usando, cuál es el nombre de nuestra cuenta, etc., etc. Son las llamadas **variables de entorno**.

Otras variables de entorno son utilizadas por otra clase de programas para sus propios fines. El usuario también puede definir sus propias variables de entorno.

Las variables de entorno son globales o **exportadas**, esto es, pueden ser accedidas por los procesos lanzados por el *shell*. Se pueden definir parámetros de otro tipo, las **variables del *shell***, visibles sólo localmente dentro del *shell*.

Para ilustrar qué son las variables, empezaremos definiendo una variable nosotros mismos. La forma de hacerlo depende del *shell*. Si trabajamos con el **sh** o el **ksh**, teclearemos

```
mi_variable=hola
```

Si utilizamos el **csh**, habría que escribir

```
set mi_variable=hola
```

El resultado en ambos casos es que se define una variable, **mi_variable**, con el valor "hola".

Las variables de entorno también distinguen mayúsculas, es decir que la variable **MI_VARIABLE** no es la misma que **mi_variable**. Las variables de entorno predefinidas por el *shell* suelen ir en mayúsculas.

11.6.1 La orden `echo`

La orden interna **echo** imprime el texto que se le escriba a continuación. Con ella se puede visualizar el contenido de parámetros del *shell* si se escribe el nombre del parámetro precedido por **$**. Ejemplo:

```
$ echo la variable es $mi_variable
la variable es hola
$
```

Los parámetros pueden ir en medio de una cadena, siempre que el carácter que venga a continuación del parámetro sea un **separador** (espacio, coma, punto y

coma...). Si se desea intercalar un parámetro dentro de una palabra más grande, hay que escribir su nombre entre llaves, como en el ejemplo:

```
$ echo c$mi_variable
chola
$ echo coc$mi_variablelate
ksh: mivariablelate: parameter not set
$ echo coc${mi_variable}late
cocholate
$
```

En el segundo caso, el *shell* intenta imprimir la variable **mi_variablelate**, que en este ejemplo no está definida y por tanto no escribe nada o nos da error.

11.6.2 Sustitución de parámetros

Los parámetros del *shell* no se sustituyen sólo mediante **echo**. En realidad, el *shell*, *antes* de ejecutar una orden, detecta todas las cadenas del tipo **$nombre** y las intenta sustituir por el contenido de parámetros. A continuación se ejecuta la orden.

Por ello, las variables de entorno se pueden escribir dentro de cualquier orden:

```
$ ls
pepe.c
pepe.x
un_fichero
$ fich=un_fichero
$ ls $fich
un_fichero
```

La sustitución de nombres de ficheros con comodines se realiza *después* de la sustitución de variable:

```
$ ls
hola.c
pepe.c
pepe.x

$ c="*.c"

$ ls $c
hola.c
pepe.c
```

En este ejemplo, se sustituye el parámetro **c** por la cadena ***.c**, resultando la orden **ls *.c**

11.6.3 La orden `set`

Describiremos ahora cómo listar los parámetros del *shell* disponibles. La orden interna **set** lista todos los parámetros del *shell*, incluyendo variables de entorno y variables del *shell*. La lista puede ser algo parecido a esto:

```
$ set
HOME=/users/pepe
LINENO=1
LINES=24
LOGNAME=pepe
MAIL=/usr/mail/pepe
MANPATH=/usr/man:/usr/contrib/man:/usr/local/man
PATH=/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin:.
PPID=28552
PS1=$SYSTEM:
PS2=>
PS3=#?
PS4=+
PWD=/users/pepe/practica
SHELL=/bin/ksh
SYSTEM=sopa
TERM=vt100
TMOUT=7200
TZ=GMT0BST
```

Algunos parámetros de interés son HOME, PATH, SHELL, etc., que son variables de entorno empleadas por el *shell* para conocer el directorio local del usuario, dónde buscar ficheros ejecutables, etc. En el apartado XXX se listan unas cuantas variables de entorno interesantes.

Muchos parámetros están puestos a punto por el propio *shell*, y en bastantes casos es mejor no alterarlos, o incluso es inútil.

En el procesador de órdenes **sh**, las variables de entorno suelen ir en mayúsculas, mientras que en el **csh** sus equivalentes van en minúsculas: **home, path**, etc.

11.6.4 Variables de entorno

Un parámetro definido en un *shell* no es en principio "heredado" por los programas que ejecuta.

Para que un parámetro sea parte del entorno del *shell*, hay que declarar que se va a exportar, con la orden **export**:

```
$ TEMP=$HOME/tmp
$ export TEMP
```

En este ejemplo se declara la variable **TEMP** como exportada, o sea, se convierte en variable de entorno y todos los programas lanzados por el *shell* podrán utilizarla.

La orden **env** lista los parámetros del *shell* heredadas por el *shell*; son las variables de entorno.

11.6.5 Borrado de parámetros

Finalmente, para borrar variables de entorno se dispone de las orden interna **unset**:

```
unset mi_variable
```

En el **csh**, cuando se trata de borrar una variable exportada, hay que teclear:

```
unsetenv mi_variable_global
```

11.6.6 Rutas de búsqueda: la variable `PATH`

Cuando se escribe una orden, en general se lee un fichero ejecutable y se ejecuta. ¿Cuál es ese fichero ejecutable? Si por ejemplo se teclea **ls**, se ejecutará un fichero-programa llamado **ls**, situado en algún punto del sistema de ficheros (en algún directorio como **/bin** o **/usr/bin**).

¿Cómo sabe el *shell* en qué directorio se halla el programa para ejecutar? Existe un parámetro de nombre **PATH** donde se declaran las posibles ubicaciones de los ficheros ejecutables. El formato de PATH es una serie de rutas separadas por el carácter dos puntos **:**.

En UNIX bastantes programas del sistema se hallan en los directorios **/bin** y **/usr/bin**. La variable PATH podría tener este valor:

```
PATH=/bin:/usr/bin
```

En este caso, si existe el fichero **/bin/ls**, basta entonces con escribir **ls** para que se ejecute. El procesador de órdenes busca en los directorios declarados en **PATH** hasta encontrarlo.

Si existiesen tanto **/bin/ls** como **/usr/bin/ls**, se ejecutaría **/bin/ls**, que es el primero de la lista de **PATH**.

La variable **PATH** sólo sirve para búsquedas de ficheros ejecutables, no de ficheros de datos.

Como práctica sobre las variables de entorno, demos dos ejemplos relacionados con **PATH**:

Para visualizar las rutas de búsqueda, se puede entrar:

```
echo $PATH
```

Para añadir el directorio **/usr/local/bin** a las rutas de búsqueda:

```
PATH=$PATH:/usr/local/bin
```

En este ejemplo, el directorio **/usr/local/bin** será el último en explorarse.

11.6.7 Algunas variables de entorno

HOME	Directorio por omisión para **cd**; normalmente el directorio local
PATH	Rutas de búsqueda de órdenes
PS1	Texto (*prompt*) primario del *shell*. Por defecto es **$**.
PS2	Texto (*prompt*) secundario del *shell*, para continuación de líneas. Por defecto es **"> "**
PWD	Directorio de trabajo actual
SHELL	Ruta completa del *shell*. El programa que se ejecuta para interpretar escritos de *shell*.
TMOUT	Si este parámetro es un número positivo, si no se entra una orden o INTRO en **$TMOUT** segundos, el *shell* termina por sí solo.
VISUAL	Si tiene el valor **emacs**, **gmacs** o **vi**, se adopta el correspondiente modo de edición interactiva de órdenes.

11.7 Sustitución de órdenes y comillas

11.7.1 Sustitución de órdenes

El **ksh** permite reemplazar una orden por su resultado escrito, dentro de una línea de órdenes. Existen dos métodos para ello:

1. Escribir la orden entre acentos graves: **`orden`**
2. Escribir un dólar y la orden entre paréntesis: **$(orden)**

Por legibilidad, el método recomendado es el segundo.

Viéndolo con un ejemplo:

```
$ echo el directorio actual es `pwd`
el directorio actual es /home/pepe
$ echo el directorio actual es $(pwd)
el directorio actual es /home/pepe
```

En ambos casos, la orden **pwd** se ejecuta y su resultado es *expandido* en el texto original.

En la expansión los caracteres de nueva línea se convierten en espacios. Es decir que todo el texto va en una sola línea.

```
$ ls
pepe.txt
juan.dat
$ echo "Los ficheros son " $(ls)
Los ficheros son pepe.txt juan.dat
```

Si la orden sustituida va dentro de un texto entre comillas, los saltos de línea se preservan:

```
$ echo "Los ficheros son $(ls)"
Los ficheros son pepe.txt
juan.dat
$
```

Dentro de los paréntesis puede ir cualquier orden, con redirecciones, conductos, parámetros, etc., e incluso otras sustituciones de órdenes.

Supóngase que cierto fichero **borrables** contiene una lista de ficheros que se van a borrar. Esto se puede conseguir con la siguiente orden:

```
rm $(cat borrables)
```

dado que la lista de los ficheros se expande tras el **rm**. El mismo efecto se puede lograr en el **ksh** con

```
rm $(<borrables)
```

que es más rápido, porque no se llama al programa **cat**.

11.7.2 Pasos en la ejecución de una orden

Estamos en situación de enumerar los pasos que se realizan para ejecutar una orden, una vez que ha sido entrada. Recordar en qué orden se efectúan las sustituciones es importante para saber cuándo es necesario dejar textos entre comillas.

1. Se sustituyen los parámetros del *shell*
2. Se expanden los nombres de ficheros (*wildcards*, etc.)
3. Si es una orden interna, se ejecuta. Si no, se busca el fichero ejecutable en las rutas declaradas en **$PATH**
4. Si se encuentra el fichero, se ejecuta

11.7.3 Entrecomillado

Hay ocasiones en las que interesa que el *shell* no desarrolle ni sustituciones de nombres de ficheros, ni de nombres de variables, ni nada parecido. Estos tres mecanismos evitan interpretaciones indeseadas:

11.7.4 Sintaxis	Ejemplos	Función
\carácter	find / -name *.c	Interpreta el carácter literalmente
'texto'	echo '$PATH' echo 'mira la barra \'	Interpreta el texto literalmente
"texto"	echo "ruta=$PATH"	Sólo expande variables y sustitución de órdenes

La **barra invertida** o *backslash* sirve para interpretar literalmente el carácter situado a su derecha. Así se evitan expansiones de variables, de nombres de ficheros, comillas, etc.

```
$ echo "Esta letra \" es una comilla"
Esta letra " es una comilla
```

Sobre un texto entre **comillas simples** el *shell* no realiza ninguún tipo de sustitución, toma el texto literalmente:

```
$ echo '$user'
$user
$ echo '*.c'
*.c
$ echo '`cd`'
`cd`
```

© *José Miguel Santos Espino*

Sobre un texto entre **comillas dobles**, el *shell* expande las variables y realiza sustituciones de órdenes. No interpreta los asteriscos ni demás expresiones de expansión de nombres. La barra invertida se puede usar para interpretar literalmente los caracteres ****, ´, **"** y **$**.

```
$ echo "$USER"
pepe

$ echo "\$USER"
$USER

$ echo "*.c"
*.c

$ echo "`pwd`"
/users/pepe
```

Otro ejemplo; la orden

```
$ ls "<salida" "*asterisco*"
```

lista dos ficheros que se llaman respectivamente **<salida** y ***asterisco***, cuyos nombres contienen caracteres (el "menor que" y el asterisco) que podrían resultar conflicitvos si no se escribieran entre comillas.

Un texto entre comillas dobles es considerado un único argumento. Así, la orden

```
$ ls "mi fichero"
```

intenta listar un fichero llamado **mi fichero**, cuyo nombre tiene un espacio intercalado.

11.8 Edición interactiva de órdenes

Los procesadores de órdenes en UNIX no se han distinguido precisamente por su amigabilidad a la hora de escribir órdenes. Sin embargo, con los años se han ido aportando herramientas para editar órdenes interactivamente, e incluso llevar un registro de órdenes ya lanzadas que se puedan volver a invocar pulsando sólo un par de teclas.

El **csh**, el **ksh** y sus herederos disponen de edición interactiva e historial de órdenes. En este apartado se tratará de esas facilidades en el **ksh**, casi idénticas al *shell* POSIX.

11.8.1 Completar nombres de ficheros

Esta función, llamada en inglés *file name completion*, consiste en que, escribiendo sólo el comienzo de un nombre de fichero, y pulsando a continuación dos veces ESC, se rellena el resto del nombre, si es que hay algún fichero que encaja con lo ya escrito.

Por ejemplo, si en el directorio actual hay un fichero llamado **fichero_de_largo_nombre**, y se teclea:

$ getaccess fich<ESC> <ESC>

aparecerá en pantalla:

$ getaccess fichero_de_largo_nombre

Hasta que no se pulse INTRO, no se cursará la orden. Así se puede seguir tecleando más texto tras el nombre de fichero expandido.

En caso de que existan varios ficheros que encajen con el prefijo indicado, se expande el nombre hasta el prefijo común más largo de todos esos ficheros. Con un ejemplo, si existieran tres ficheros llamados **fichero1**, **fichero2** y **fichero.txt**, al teclear:

$ ls fich<ESC><ESC>

aparecería

$ ls fichero

A veces es conveniente expandir *todos* los nombres que se ajusten al mismo prefijo. Esto se logra con la combinación <ESC>*, escape y asterisco. En el ejemplo anterior, tecleando

$ ls fich<ESC>*

se obtendrá

$ ls fichero1 fichero2 fichero.txt

El completar nombres también se aplica a rutas que incluyan directorios.

11.8.2 Expansión de variables

El mismo mecanismo para completar nombres de ficheros se utiliza para expandir interactivamente el valor de una variable. Si se teclea <ESC><ESC> tras un nombre de variable, se cambia por su contenido. Ejemplo:

$ ls $HOME<ESC><ESC>

da lugar a algo similar a

$ ls /users/pepe

11.8.3 Modos de edición interactiva

Los procesadores de órdenes UNIX más tradicionales no daban prácticamente facilidades para la edición de órdenes. Simplemente se escribían caracteres, y todo lo más se podían borrar los últimas letras hacia la izquierda. El **ksh** y otros *shells* más modernos incorporan la posibilidad de editar la línea de órdenes, con funciones de ir al inicio de la línea, al final, insertar texto, etc.

Existen dos modos principales de edición: el "vi" y el "emacs", cada uno inspirado en órdenes de un editor de texto distinto. En este apartado se describirá el modo de edición "emacs", el cual, aunque quizás menos potente, es más fácil de aprender y de utilizar.

El modo de edición tiene que ser activado, lo cual puede hacerse de varias formas. Por ejemplo, para pasarse al modo de edición "emacs" puede teclearse

```
set -o emacs
```

o bien

```
VISUAL=emacs
export VISUAL
```

orden que puede dejarse en el fichero **.profile** de nuestro directorio local, para configurarlo de forma permanente.

Las órdenes del modo de edición "emacs" son combinaciones <CTRL>+***carácter***, o <ESC>***carácter***. A continuación se listan algunas de las combinaciones más usuales. El significado de las combinaciones es este:

^***letra*** indica pulsar la tecla "control" simultáneamente con la tecla ***letra***

<ESC>*letra* indica pulsar la tecla de "escape" y posteriormente la ***letra***

^B	Cursor un carácter a la izquierda (*Backward*)
^F	Cursor un carácter a la derecha (*Forward*)
^A	Cursor al principio de línea
^E	Cursor al final de la línea (*End*)
<ESC>B	Cursor una palabra a la izquierda
<ESC>F	Cursor una palabra a la derecha

^D	Borra el carácter donde está el cursor
^K	Borra hasta fin de línea

11.8.4 Historial de órdenes

Los *shells* de UNIX mantienen en un fichero o en memoria interna el historial de las últimas órdenes lanzadas: es el **historial de órdenes** (*command history*). Este registro histórico puede utilizarse para repetir una orden escrita hace poco (situación muy frecuente) o modificar levemente una orden ya lanzada (situación no menos frecuente).

Cada vez que se escribe cualquier línea no vacía y se pulsa INTRO, queda ingresada en el historial de órdenes. El historial conserva la memoria de pasadas sesiones con el *shell*.

Para emplear el historial desde el modo de edición "emacs", pueden usarse estas combinaciones de teclas:

^P	Retrocede a la orden anterior (*Previous*)
^N	Avanza en el historial (*Next*)
^R*cadena*	Busca la orden más reciente donde aparezca el texto *cadena*

Relacionada con el historial de órdenes, existe otra combinación de tecleas bastante práctica:

<ESC>	Añade a la línea de órdenes la última palabra de la orden anteriormente tecleada

Con un ejemplo:

```
$ ll pepe.c
-rw-rw-rw-  1 jomis   users     62 Nov 28 12:17 pepe.c
$ rm <ESC>.
```

aparece

```
$ rm pepe.c
```

11.9 Otras características del *shell*

A continuación se describen algunas características misceláneas del *shell*.

11.9.1 Concatenación de órdenes

Si se desea especificar múltiples órdenes en la misma línea, sepárelas por punto y coma:

```
cd; ls
```

Las órdenes se ejecutan secuencialmente.

11.9.2 Múltiples procesos concurrentes

Mediante la sintaxis

```
c1 & c2 & c3
```

o sea, órdenes separadas por ampersands, se lanzan dichas órdenes concurrentemente.

11.9.3 Continuación de líneas

Si vemos que la línea de órdenes está demasiado llena, se puede proseguir en la siguiente línea de forma elegante escribiendo la barra invertida **** seguida de INTRO.

El *shell* responde con un *prompt* especial de "segunda línea", habitualmente un **>**, pudiéndose continuar la orden en la siguiente línea.

```
$ find / -name "*.c" -print \
> >lista.programas.en.c
$
```

11.9.4 Alias

Una forma de abreviar órdenes más eficiente que las variables de entorno o los escritos (los cuales trataremos a continuación) es usar la orden interna **alias**, con la sintaxis

alias *nombre_del_alias=cadena sustituida*

Si se declara un alias, y el *shell* localiza el nombre de ese alias en una línea de órdenes, sustituye ese nombre por la cadena de sustitución. Con un ejemplo:

```
$ alias lld="ls -ld"
$ lld
```

```
-rwxr-xr-x  1 pepe       18 Jun 17 12:41 fich
-rw-r--r--  1 pepe      138 Jun 16 11:14 foch
$
```

Nótese que si la cadena contiene caracteres especiales, es necesario entrecomillarla.

Si tecleamos **alias** a secas, se nos ofrece la lista de los alias definidos.

Para borrar un alias, teclee

unalias *nombre_del_alias*

11.9.5 Salida del *shell*

La orden interna **exit** finaliza el procesador de órdenes. Si era el *shell* con el que ingresamos (*login shell*), además abandonaremos la sesión.

Tecleando ^D, indicamos al *shell* que se ha terminado la entrada estándar y también finaliza. Pero es preferible usar **exit**.

Para realmente asegurarnos de salir del sistema, es mejor teclear **logout** en el **csh**.

12. Utilidades de manipulación de ficheros

En este apartado se describen tareas típicas de creación, visualización, copia o borrado de ficheros bajo UNIX, que a buen seguro van a ser moneda corriente en sus largas jornadas de trabajo.

12.1 ls: Información sobre ficheros

El programa **ls** lista el contenido de directorios, o imprime información diversa sobre un conjunto de ficheros.

Algunas opciones interesantes son:

-l	listado largo. Aparece, para cada fichero, los permisos, el propietario, la longitud, fecha y hora de última modificación y el nombre.
-a	lista todos los ficheros, incluidos aquellos que empiezan por un punto.
-d	Si el argumento es un directorio, no lista su contenido, sino su nombre. Se suele usar junto con la opción l.
-F	Añade ciertos caracteres para distinguir los ficheros ejecutables (*), los directorios (/) y otros tipos de ficheros.
-R	Listado recursivo. Se lista también el contenido de los subdirectorios.
-t	Ordena la lista según el tiempo de última modificación (los más recientes, primero)

12.2 Visualización de ficheros

La orden

cat fichero

sirve para ver por pantalla el contenido de un fichero. Si el fichero es muy extenso, tendrán que hacer uso de ^S y ^Q para controlar el flujo de los caracteres.

El programa **cat** admite la sintaxis

cat fichero1 fichero2 … ficheroN

para visualizar el conjunto de ficheros especificados. Como se ha visto, este programa puede utilizarse con redirección para generar nuevos ficheros.

Para no preocuparse por los ficheros que ocupan más de una pantalla, se puede recurrir al programa **more**, con esta sintaxis:

more [opciones] ficheros

Algunas de las opciones reconocidas por **more** son

+num_lin visualiza a partir de la línea **num_lin**

-r visualiza también los caracteres de control

12.3 cp: copia de ficheros

El programa **cp** admite dos variantes:

cp fichero_fuente fichero_destino

para copiar un fichero en otro, y

cp fich1 fich2 ... fichN directorio

para copiar un conjunto de ficheros en un directorio.

12.4 mv: renombrado o movimiento de ficheros

Teclee

mv fichero1 fichero2

y renombrará **fichero1** como **fichero2**. Si **fichero2** es una ruta que termina en otro directorio, el efecto de la orden será mover el fichero a ese directorio.

Para mover un grupo de ficheros hacia un directorio:

mv fichero1 fichero2 ... directorio

12.5 rm: borrado de ficheros

Para borrar uno o más ficheros, se emplea el programa **rm**.

rm fich1 fich2 ... fichN

Si se desea borrar un directorio recursivamente, utilice

rm -r dir1 dir2 ...

Recuerden que en UNIX no se pueden recuperar ficheros, así que utilicen la opción **-r** con prudencia.

La opción **-i** solicita interactivamente al usuario confirmación del borrado, de cada uno de los ficheros implicados.

12.6 Creación de enlaces

El UNIX permite la creación de múltiples entradas en el sistema de ficheros que apunten todas ellas a un solo fichero.

Supongamos que queremos que varios usuarios dispongan en su directorio local de un fichero con el mensaje del día. Una opción es que cada día tal fichero se copie en el directorio local de todos los usuarios, lo cual representa un gasto de espacio, así como estar ejecutando múltiples copias.

En UNIX se pueden crear entradas de directorio (vamos, ficheros) que conecten con otros ficheros ya existentes, dando la apariencia de que el mismo fichero está duplicado en varios puntos del árbol de directorios. A todos estos "enganches" hacia un fichero se les denomina **enlaces** (*links*).

Para crear un enlace, se utiliza la orden **ln**, según esta sintaxis:

ln fichero.original fichero.enlace

El nuevo fichero será idéntico al original, conservará su mismo contenido, su propietario será el mismo, los permisos serán iguales... lo único que se ha construido es una nueva forma de nombrar a la misma entidad. Por tanto, si se modifica el contenido de un fichero o enlace, repercute automáticamente en los restantes enlaces.

Ahora bien, si se borra el fichero original o cualquier enlace, no se borran los restantes enlaces. El fichero sólo desaparece cuando el último enlace que apunte a él se borre. Se permite borrar un fichero aunque los permisos lo denieguen, siempre que exista otro enlace apuntando a él.

12.6.1 Enlaces simbólicos

En UNIX no se permite establecer enlaces entre sistemas de ficheros distintos. Esto significa que en la mayoría de los sistemas no se puede crear un enlace desde / a /usr. Para ello existe una herramienta similar, que son los **enlaces simbólicos**.

Para establecer un enlace simbólico, añada la opción **-s** en una orden **ln**.

Los enlaces simbólicos difieren en comportamiento respecto a los ordinarios (a los que se llama enlaces "duros").

13. El editor vi

El sistema operativo UNIX incorpora varios editores estándares, uno de los cuales está orientado a terminales de caracteres. Es el llamado **vi**, al cual dedicaremos todo este capítulo.

El editor de texto **vi** no se caracteriza precisamente por su amigabilidad; en los entornos de ventanas, como el VUE, encontraremos editores de texto dirigidos por ratón mucho más sencillos de usar. En entornos no gráficos, hace tiempo que se utiliza otro editor de la GNU, el **emacs**, también menos tortuoso que el **vi**. El editor **vi** mantiene las ventajas de que es el editor estándar de UNIX y que es bastante potente (en el sentido de que ofrece un sinfín de operaciones).

Un operador UNIX que se precie debe conocer el **vi**, porque más tarde o más temprano se verá obligado a recurrir a él. Una situación típica puede ser al arrancar la máquina en modo monousuario sin entorno gráfico, donde no disponemos de los cómodos editores gráficos.

13.1 Modo de órdenes, modo de inserción

El editor **vi** es un programa más de UNIX, que se invoca con o sin parámetros. Si se le invoca de esta forma

vi *fichero*

aparecen en pantalla las primeras líneas del *fichero* y entramos a editar. Si se invoca sin parámetros, empezamos con un fichero nuevo (al que más tarde habrá que dar nombre).

Si el fichero tiene menos líneas que la pantalla de la terminal, o si estamos con uno nuevo, el editor rellena las líneas de pantalla con el carácter tilde ~

Ejemplo: **vi nuevo.txt**

Si **nuevo.txt** no existiera, aparecería en pantalla algo como lo siguiente:

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"nuevo.txt" [New file]
```

Una vez se ha entrado en el editor, se puede empezar a manipular el fichero. Pero el **vi** no nos va a dejar escribir caracteres alegremente; si tecleamos cualquier letra, veremos que, como mínimo, no aparece nada en pantalla. Esto es porque el editor

se encuentra inicialmente en el **modo de órdenes**, en el cual cada letra representa una orden al editor. Por ejemplo, una 'x' es borrar el carácter encima del cursor, una 'j' es desplazar el cursor una línea hacia abajo, etc.

Para entrar nuevo texto hay que pasar del modo de órdenes al **modo de inserción**, en el que las letras se escriben en el texto. Algunas órdenes del **vi** pasan a modo de inserción, como 'a', que significa "empieza a insertar texto a partir del cursor".

Cuando se termina de introducir texto y se desea volver al modo de órdenes, se pulsa la tecla <ESC> (escape). Con un ejemplo: llame al editor y pulse una "a". Podrá introducir texto a placer; para terminar pulse <ESC>.

Algo que observará es que normalmente no puede enmendar los fallos que ha tenido tecleando, y a lo mejor no se puede desplazar por el texto usando los cursores. Y es que para realizar esas actividades, normalmente hay que retornar al modo de órdenes.

El editor nunca informa visualmente si estamos en modo de inserción o de órdenes. Lo único es que pulsando <ESC> estando en modo de órdenes, suena un pitido de confirmación.

Otra consideración es que el **vi** distingue mayúsculas de minúsculas: la orden 'h' retrocede un espacio en el texto, mientras que 'H' lleva el cursor a la esquina superior izquierda de la pantalla.

A continuación daremos una escueta visita a los distintos tipos de órdenes del editor **vi**. Para una referencia rápida, puede consultar la tabla-resumen de órdenes al final de este capítulo.

13.2 Tipos de órdenes

Como ya indicamos, podemos llamar al editor **vi** con un argumento que es el fichero que se va a editar, como por ejemplo

```
vi nuevo.txt
```

Invoque esta orden. Le aparecerá la pantalla de edición, con la línea inferior reservada para los mensajes del editor.

Para introducir nuevo texto, recuerde que tiene que pasar al modo de inserción. Para ello puede pulsar

- **a** para insertar texto a la derecha del cursor
- **i** para insertar texto a la izquierda del cursor

En el modo de inserción no se pueden realizar movimientos complejos de cursor ni otras operaciones; no obstante, muchos sistemas vienen con unas macros que permiten utilizar las flechas de cursor del teclado para moverse incluso en modo de inserción, como casi seguro ocurre en el sistema que usted usa. Pero tenga presente que esto no lo vamos a encontrar siempre.

Para volver al modo de órdenes, hay que pulsar <ESC>.

Si se quiere guardar el fichero, estando en modo de órdenes se teclea **:w**

Si se quiere abandonar el editor guardando los cambios, se teclea **:x**

Si se quiere abandonar el editor sin guardar los cambios, se teclea **:q!**

El editor **vi** dispone de multitud de funciones para moverse a lo largo del fichero y manipularlo. En este sentido es bastante potente, aunque su potencia no va acompañada de facilidad de uso. Podríamos clasificar las órdenes del **vi** en estas categorías:

- Movimiento del cursor
- Inserción de texto
- Borrado de texto
- Manipulación del búfer
- Búsquedas y sustituciones de texto
- Acciones sobre el fichero
- Configuración del editor

Los tres primeros tipos de órdenes, movimiento del cursor, inserción y borrado de texto, resultan más o menos evidentes. El editor **vi** dispone de un *búfer* en memoria donde puede guardarse un texto para luego depositarlo en otro punto del fichero, duplicarlo, etc. Se trata ni más ni menos que las conocidas operaciones de cortar-copiar-pegar.

Existen órdenes bastante versátiles para realizar búsqueda y sustitución de cadenas en el texto.

El fichero en edición se puede guardar, se puede archivar en otro fichero; también se puede insertar en el texto el contenido de otro fichero, etc. Todas las órdenes de este tipo comienzan por el carácter dos puntos ':'

También comienzan por ':' las órdenes encaminadas a configurar el editor. Por ejemplo, para hacer que aparezca a la izquierda de cada línea su número relativo, se escribe la orden **:set number**.

13.3 Resumen de órdenes del vi

Esta tabla describe algunas de las más importantes órdenes del **vi**, clasificadas según su función. Sólo es posible utilizarlas en modo de órdenes. Algunas de las órdenes pasan a modo de inserción, lo cual queda indicado en la descripción de las mismas.

Se adoptan las siguiente convenciones:

N	es un número, por ejemplo **124**.
c	es un carácter cualquiera, por ejemplo **a**.
s	es una cadena de caracteres, por ejemplo **hola**. Puede contener espacios.

Movimiento del cursor

h	retrocede un espacio
***N*h**	retrocede *N* espacios
l	avanza un espacio
j	avanza una línea
k	retrocede una línea
w	avanza una palabra
***N*w**	avanza *N* palabras
b	retrocede una palabra
e	va al final de la siguiente palabra
^	va al inicio de la línea actual
$	va al final de la línea actual
H	va al principio de la pantalla
M	va al centro de la pantalla
L	va a la última línea de la pantalla
^B	retrocede una página
^F	avanza una página
***N*G**	va a la línea *N*
G	va a la última línea del fichero

Borrado de texto

x	borra el carácter sobre el cursor
***N*x**	borra *N* caracteres a partir del cursor
dw	borra la palabra a la derecha del cursor
db	borra la palabra a la izquierda del cursor
d*N*b	borra *N* palabras a la izquierda del cursor
dd	borra la línea actual
d$	borra hasta fin de línea

NOTA: las órdenes de borrado guardan el texto suprimido en el *búfer* del editor.

Edición de texto

J	funde dos líneas consecutivas en una sola
u	deshace la última operación del editor
r*c*	reemplaza el carácter encima del cursor por *c*
cw*s*<ESC>	reemplaza la palabra encima del cursor por la cadena *s*

Inserción de texto (pasan a modo de inserción)

i	inserta texto antes del cursor

a	inserta texto a partir del cursor
I	inserta texto desde el principio de la línea actual
A	inserta texto desde el final de la línea actual
o	abre una línea bajo el cursor y pasa a modo inserción
O	abre una línea sobre el cursor y pasa a modo inserción
<ESC>	en modo de inserción: pasa a modo de órdenes en modo de órdenes: emite un sonido

Búsqueda y sustitución de cadenas

/*s*	Busca *s* hacia adelante en el fichero
?*s*	Busca *s* hacia atrás en el fichero
//	Repite la última búsqueda
:1,$s/*s1*/*s2*/g	Sustituye la cadena *s1* por la cadena *s2* en todo el fichero
:3,8s/*s1*/*s2*/g	Sustituye la cadena *s1* por la cadena *s2* desde la línea 3 hasta la línea 8
:.,10s/*s1*//g	Borra las apariciones de la cadena *s1* desde la línea actual hasta la línea 10
:1,.s/*s1*/*s2*/	Sustituye la primera aparición de *s1* por *s2*, buscando desde la línea 1 hasta la actual

Manejo del búfer

yw	Copia la siguiente palabra en el búfer
y*N*w	Copia las siguientes *N* palabras en el búfer
yy	Copia la línea actual en el búfer
***N*yy**	Copia *N* líneas a partir de la actual en el búfer
p	Vuelca el contenido del búfer a la izquierda del cursor (operación de pegar)
P	Vuelca el contenido del búfer a la derecha del cursor (operación de pegar)

NOTA: las órdenes de borrado depositan el texto suprimido en el búfer (operación de cortar)

Ficheros

:w	Almacena los cambios del texto en el fichero
:w *fich*	Escribe el texto del editor en el fichero *fich*
:r *fich*	Deposita el contenido de *fich* a la derecha del cursor

Otras órdenes

:q	Abandona el editor
:q!	Abandona el editor, sin guardar los cambios
:x	Abandona el editor, guardando los cambios
:!*orden*	Ejecuta la *orden* del shell (p.ej. :!ls)
:set	Muestra las opciones activas del editor
:set *opción*	Activa la *opción* del editor

Algunas opciones

autoindent: al abrir una nueva línea, conserva el sangrado de la anterior

number: muestra los números de línea

A pesar de la aparente complejidad, hay una estructura común entre todas las órdenes del vi. Por ejemplo, para repetir una orden varias veces, se escribe un número y luego la orden. Para borrar hasta un punto determinado del documento, se escribe "d" y luego la orden para llegar a ese punto... con algo de paciencia y tiempo, se puede llegar a dominar un editor poco amigable, pero imprescindible en UNIX.

14. Utilidades generales

Las que vienen a continuación son utilidades no dedicadas específicamente a manipular ficheros, sino a labores específicas como imprimir, matar procesos, buscar ficheros, etc.

En algunos casos la explicación del programa sirve de excusa para comentar características generales del UNIX.

14.1 Obtención de ayuda

Dentro de una sesión de trabajo en UNIX, pueden consultar los manuales de todos los programas y órdenes del sistema, del formato de los ficheros de administración, de las funciones de biblioteca del C, etc. De ello se encarga la utilidad **man**. Entren

man ***orden***

y se les presentarán las páginas del manual referidas a **orden**. El texto estará debidamente formateado y se controla automáticamente el paso de pantallas del mismo modo que el programa **more** (de hecho, **man** hace uso de ese programa u otro similar).

A efectos de localizar una ayuda, cada orden o palabra clave que le pasen al **man** está clsificada con un número de **sección**. Por eso en las ayudas es frecuente ver remisiones a otros programas de esta manera:

... use the Bourne shell sh(1)to call this blah blah...

El (1) al lado de **sh** indica que pertenece a la sección 1. Para empezar, esto ayuda a saber de qué va el programa o fichero. Por ejemplo, la sección 1 consta de los programas del sistema de uso general. La sección sirve también para localizar la información en los manuales escritos. Y por último para distinguir entradas repetidas: por ejemplo, existen tanto **passwd(1)**, ayuda sobre el programa **passwd**, como **passwd(5)**, ayuda sobre el formato del fichero **/etc/passwd**. Si deseamos consultar una ayuda sobre una entrada en una sección concreta, se usa esta sintaxis:

man *sección orden*

Por ejemplo:

man 4 passwd

También existen secciones como 1M, 3C, etc., compuestas de un número más una letra.

A veces no se sabe exactamente el nombre de la orden sobre la que pedir ayuda. Por ejemplo, sabemos que hay una orden para cambiar la contraseña (*password*, en inglés), pero no nos acordamos del nombre.

En estos casos se recurre a la opción **-k** del man:

man -k password

que buscará todas las entradas de páginas del manual cuyos títulos contengan la palabra "password".

14.2 Impresión de ficheros

En UNIX no se puede trabajar directamente con el dispositivo impresor a no ser que se sea superusuario, e incluso en este caso es poco recomendable. La impresión de documentos se realiza en este sistema a través de un mecanismo de *spooling*, que a grandes rasgos consiste en que los ficheros para imprimir se depositan en un directorio especial llamado **directorio de *spool***.

Un programa del sistema, llamado *demonio impresor*, verifica cuándo la impresora está disponible y vigila periódicamente el directorio de *spool* en busca de nuevos trabajos. El demonio impresor es quien selecciona y envía los documentos a la impresora.

A nivel del usuario, se imprime mediante los programas **lp** o **lpr**. Tecleen

lp ***fich1 fich2 ... fichN***

y sus ficheros se enviarán al directorio de *spool* . Más tarde o más temprano saldrán impresos por la impresora, si no hay problemas.

Obsérvese que en UNIX la impresión se realiza fuera de línea, para evitar retardos y competencia entre distintos usuarios que pretendan trabajar simultáneamente con la impresora.

14.3 stty: configuración de la terminal

En UNIX, los procesos se ejecutan adscritos a una terminal. Una terminal puede tener entidad física (una consola con teclado y pantalla) o ser una terminal virtual, como cuando abrimos una ventana o efectuamos una conexión remota a través de un módem.

El UNIX define una serie de parámetros de funcionamiento de la terminal, por ejemplo la velocidad de transmisión, la paridad que emplea, etc. También se definen unas combinaciones de teclas para que produzcan una serie de acciones. Por ejemplo, en algún sitio está declarado que pulsando ^C se aborta la ejecución del proceso en curso.

Todos esos parámetros se pueden definir o alterar con el programa **stty**. Tecleando **stty** se muestra una parte de la configuración. Con **stty -a** se muestra la configuración completa. Con **stty all** se muestra sólo la configuración del teclado.

La configuración del teclado se realiza asignando una combinación de teclas a una acción. Algunas de las acciones y sus teclas habitualmente asociadas son

erase	borrar un carácter (^H)
werase	borrar una palabra (^W)
susp	dejar el proceso en segundo plano (^Z)

intr	abortar el proceso (^C)
stop	detener el flujo de caracteres (^S)
start	reanudar el flujo de caracteres (^Q)
eof	fin de datos (^D)

Algunas de ellas son viejas conocidas nuestras a estas alturas. Para modificar una asignación tecla/acción, se emplea la sintaxis

stty ***acción tecla***

Por ejemplo, para forzar el aborto de un proceso con ^G, escriba tal cual

```
stty intr ^G
```

La orden **tty** informa de cuál es el dispositivo que se emplea como terminal actualmente. Como todos los dispositivos son ficheros, se puede escribir directamente sobre una terminal, por ejemplo con **cat**:

```
$ tty
/dev/ttyp0

$ cat >/dev/ttyp0
hola
^D
hola

$
```

14.4 Comunicación entre usuarios

En muchos sistemas UNIX existen servicios de mensajería entre usuarios de la misma máquina o del mundo exterior (siempre que estén conectados al exterior). En los entornos de ventanas, como el OPEN LOOK, se dispone de utilidades amigables para correo electrónico, de fácil aprendizaje.

14.4.1 Correo electrónico

La utilidad de correo más simple es el programa **mail**. Tecleen

mail ***usuario***

para remitir un texto a algún usuario, texto que tendrán que entrar a continuación (o pueden redirigir la entrada con **mail usuario <fichero**). El formato del texto es un número arbitrario de líneas, finalizando con una línea que comience por un punto '.' Lean el manual para conocer las opciones y formatos.

Para leer mensajes remitidos a su cuenta, tecleen simplemente **mail** y entrarán en una visualización interactiva de los mensajes. Escriban **?** para ver la ayuda.

Otras utilidades de correo más recomendables con **mailx** y, mejor aún, la aplicación **elm**, la cual ofrece una interfaz más o menos amigable, incluso con ayuda interactiva.

14.4.2 Comunicación interactiva con write

Para comunicarse interactivamente con otros usuarios activos en el sistema, existe el programa **write**. Si teclean

write ***usuario***

enviarán un texto al usuario especificado, que aparecerá directamente sobre la terminal o la ventana de *shell* donde él esté trabajando. Sólo cuando pulsen INTRO se verá la línea que han escrito en la otra terminal.

Para terminar de darle la lata al otro usuario, tecleen ^D.

Con la orden

write ***usuario <fichero***

enviarán todo el contenido de un fichero a la otra terminal. Esto a veces es más conveniente.

14.4.3 Multidifusión

Si se ven en la necesidad de enviar un mensaje a todos los usuarios presentes, utilicen el programa **wall**. Funciona igual que **write**, pero sin especificar el nombre de usuario.

14.4.4 Comunicación interactiva con talk

Otro programa más cómodo para dialogar con un usuario es **talk**. Su sintaxis es idéntica a la de **write**, pero en lugar de enviar el mensaje a la otra terminal, le solicita al otro usuario que dialogue con él. Si al otro lado lanzan el programa **talk**, se establece la comunicación por medio de ventanas en las que aparecen los caracteres que ambos usuarios van tecleando.

14.4.5 Inhabilitación de nuestra terminal

En muchas ocasiones no nos interesa recibir mensajes de otros usuarios. La orden **mesg n** inhabilita nuestra terminal como receptora de mensajes. Para volver a aceptar comunicados, hay que teclear **mesg y**.

14.5 Tratamiento de procesos

Con la orden **ps** pueden verse los procesos activos en el sistema. Si la invocamos sin opciones, lista los procesos asociados a la terminal donde se ha ejecutado esta orden.

ps admite varias opciones, algunas de las cuales son:

- **-l** da un listado largo
- **-e** lista todos los procesos, salvo los del núcleo
- **-f** listado completo: aparecen los nombres y argumentos pasados a los programas en ejecución
- **-k** lista los procesos del núcleo

Según la opción, aparecerá en cada línea el PID (identificador) del proceso, el estado (R=ejecutándose, S=en espera, T=parado...), la terminal asociada y el nombre del proceso, entre otros campos.

El PID es particularmente interesante, porque permite enviarle señales a los procesos, para lo cual se emplea la orden **kill**. Que habitualmente se utilizará para matar (forzar la terminación de) los procesos.

Para matar un proceso, teclee

kill -9 ***pid***

y se enviará una señal 9 (señal de ternimación) al proceso con identificador *pid*.

Cuando el sistema está dislocado, o tenemos una terminal colgada, habrá que acudir al programa **kill** para despejar la situación.

14.6 Búsquedas de ficheros: programa find

El programa **find** es una poderosa herramienta para localizar información dentro del sistema de ficheros. El administrador del sistema utilizará **find** con frecuencia para tareas como localizar ficheros pertenecientes a un usuario, ver los ficheros más recientes, etc.

La orden **find** a grandes rasgos permite buscar ficheros a partir de un punto en el árbol de directorios según unos criterios de búsqueda especificados en la línea de órdenes. Su sintaxis aproximada es:

find ***rutas_iniciales expresión***

Las *rutas iniciales* definen los puntos de partida por los que **find** empezará a realizar sus búsquedas, explorando recursivamente los directorios que vaya hallando a su paso.

La *expresión* indica qué ficheros buscar, si se deben imprimir, e incluso si se ejecuta alguna acción por cada fichero (como borrarlo, moverlo a otro directorio, etc.)

Algunas de las expresiones más familiares son:

-name *patrón*	busca los ficheros que encajen con un nombre o patrón
-type *tipo*	busca los ficheros que sean de un *tipo*, el cual puede ser:

- f: fichero regular
- d: directorio
- l: enlace simbólico

-size +*num*	ficheros con un tamaño mayor que *num* (en bloques de 512 bytes)
-size -*num*	ficheros con un tamaño menor que *num* (en bloques de 512 bytes)
-perm *num*	ficheros con permisos según *num* (número en octal)
-user *usuario*	ficheros pertenecientes a *usuario*
-nouser	ficheros no pertenecientes a ningún usuario registrado en /etc/passwd
-newer *fich*	ficheros más recientes que el fichero *fich*
-atime *num*	ficheros accedidos desde hace *num* días
-mtime *num*	ficheros modificados desde hace *num* días

Donde aparece *num*, puede escribirse un número con un + o un - antepuesto. El significado es:

num	Exactamente *num*
+*num*	Más que *num*
-*num*	Menos que *num*

Ciertas expresiones no sirven para buscar ficheros, sino para especificar acciones con los ficheros encontrados. Esto da una potencia enorme a **find**, pues así nos permite con una sola línea acciones tan elaboradas como borrar todos los ficheros que lleven más de diez días sin tocarse.

Por ejemplo:

-print	imprime el nombre del fichero
-exec *orden*	ejecuta una *orden*. Si se escribe en ella {}, se reemplaza por la ruta absoluta del fichero. La orden ha de terminar con \;
-ok *orden*	ejecuta una *orden* como -exec, previa confirmación del usuario

La opción **-print** habitualmente se puede omitir.

Este ejemplo simple visualiza todos los ficheros, desde el directorio raíz, con extensión .c

```
find / -name '*.c' -print
```

Obsérvese el uso de las comillas en *.c, para evitar que el *shell* las expanda antes de llamar a **find**.

Este ejemplo da un directorio completo de todos los ficheros del usuario "pepe" modificados desde hace un día.

```
find / -user pepe -mtime 1 -exec ls -ld {} \;
```

Para borrar (con confirmación del usuario) todos los ficheros colgando desde /home que superen 20Kilobytes de longitud:

```
find /home -size +40 -ok rm {} \;
```

Como esperamos que se haya apreciado, el programa **find** permite realizar tareas de cierta complejidad expresándolo con pocas palabras. El administrador del sistema ha de repasar el uso de este programa para ahorrarse tiempo de trabajo arduo delante de la terminal.

15. Utilidades para filtros

Algunos de los programas que expondremos a continuación no tienen aparentemente gran utilidad, pero combinados con otros programas mediante conductos (*pipes*) se convierten en herramientas imprescindibles en UNIX.

La descripción que damos de estos programas no es completa. Para descubrir toda su potencialidad, recomendamos la lectura de los manuales, escritos o en línea.

Como decimos, la mayoría de estos programas se pueden usar como filtros, recogiendo datos desde la entrada estándar y entregando datos a la salida estándar. Salvo que se indique lo contrario, estos programas admiten la sintaxis

programa [*opciones*] [*ficheros*] [*fich. salida*]

Si se omiten los ficheros de entrada, se toman datos de la entrada estándar. Si se omite el fichero de salida, el resultado se escribe en la salida estándar.

Combinando varios de estos programas se pueden solucionar problemas bastante elaborados en todo lo relativo a tratamiento de ficheros y de textos. De hecho, hay quien dice que UNIX es un gran procesador de textos.

15.1 Presentar las primeras o últimas líneas

head *fichero*	muestra las 10 primeras líneas de ***fichero***
head -n *num fichero*	muestra las ***num*** primeras líneas de ***fichero***
tail *fichero*	muestra las 10 últimas líneas de ***fichero***
tail -n *num fichero*	muestra las últimas ***num*** líneas de ***fichero***

15.2 Contar las líneas, los caracteres o las palabras

wc [-l -c -w] ficheros

Las opciones (que se pueden combinar) son:

-l	cuenta líneas
-w	cuenta palabras (separadas por espacios o saltos de línea)
-c	cuenta caracteres (letras)

Ejemplo:

wc -lw *.pas

Muestra el número de líneas y palabras de los ficheros de extensión **.pas**

15.3 Recortar fragmentos de líneas

Cuando interese quedarse con pedazos de líneas, el programa **cut** es de gran ayuda. Con distintas opciones se le indica que de cada línea leída sólo imprima un número de letras o de campos. Esto puede verse con algunas de las opciones que reconoce:

-c *M-N*	imprime sólo los caracteres desde el *M* hasta el *N*
	Si *N* se omite, se entiende que es hasta el final de línea
-f *M-N*	imprime sólo las palabras desde la *M* hasta la *N*
-d *C*	*C* es el carácter separador de campos (cuando se usa **-d**)

Así, si se desean borrar los dos primeros caracteres de todas las líneas de un fichero, se puede hacer con:

```
cut -c 3- fichero >fichero.cortado
```

Para quedarse sólo con los nombres de usuarios del fichero **/etc/passwd**, se puede escribir:

```
cut -d : -f 1 /etc/passwd
```

15.4 Ordenar líneas

Se emplea para ello el programa **sort**, el cual sirve también para fusionar varios ficheros de forma ordenada, o comprobar si un fichero está ordenado. Aquí, "ordenar" significa tomar las líneas de un fichero de texto y ordenarlas alfabéticamente.

Algunas de las opciones que acepta **sort** son:

-f	no distingue mayúsculas de minúsculas
-n	ordena numéricamente
-r	ordena de forma decreciente

Ejemplos:

```
sort -r <fich >ord
```

Genera un fichero ord con las líneas de **fich** ordenadas alfabéticamente en orden inverso.

```
wc -l * | sort -n
```

Muestra por la salida estándar los ficheros del directorio actual ordenados según el número de líneas que tienen.

```
ls -s | sort -nr | head -n 5
```

Muestra por la salida estándar los cinco ficheros del directorio actual que ocupan más espacio.

Si a **sort** se le pasan varios ficheros como argumentos, da como resultado una fusión ordenada de todos ellos.

De ordinario, **sort** ordena las líneas atendiendo a la línea completa; pero se le puede indicar que ordene sólo en base a un campo. Por *campos* se entienden algo así como palabras separadas por espacios.

La opción **-k *N*** obliga a **sort** a ordenar en base al campo ***N***. Con un ejemplo:

```
$ sort -n -k 3
Pepe Lotilla 1500
Pedro Medario 3870
Jacinto Lerante 132
^D
Jacinto Lerante 132
Pepe Lotilla 1500
Pedro Medario 3870
$
```

15.5 Búsqueda de patrones: grep

El programa **grep** es una herramienta inestimable para localizar cadenas de caracteres dentro de ficheros (o de la entrada estándar). Su sintaxis, en principio, es

grep *cadena ficheros*

Se buscan las líneas que contengan *cadena* en cada uno de los *ficheros*. Por ejemplo, para buscar al usuario "pepe" en el fichero /etc/passwd:

```
grep pepe /etc/passwd
```

Y aparecerá la línea (o líneas) donde figure la cadena "pepe".

grep también se puede usar como filtro, buscando cadenas en la entrada estándar:

```
who | grep pepe
```

descubre si el usuario "pepe" está activo en el sistema.

Asimismo, **grep** devuelve un valor de retorno que podemos utilizar en nuestros escritos de *shell*. Devuelve un 0 si se encontró la cadena, un 1 si no, y un 2 si hubo un error ajeno al programa.

El programa **grep** no sólo admite la búsqueda de cadenas literales, sino también patrones de caracteres un poco al estilo de los empleados en la sustitución de

nombres de ficheros en los *shells*. Sin ser exhaustivos, algunos ejemplos de patrones son:

[az]	cualquier cadena que contenga "a" o "z"
[a-z]	cualquier cadena que contenga una letra entre la "a" y la "z"
a$	una línea que termine en "a"
^a	una línea que empiece por "a"
a*	la letra "a", repetida 0 o más veces
a+	la letra "a", repetida 1 o más veces
.	cualquier carácter excepto nueva-línea

Ejemplos de patrones más elaborados:

[A-Z][A-Z0-9]*	cualquier cadena que comience por una letra mayúscula y le sigan cero o más letras mayúsculas o números.
uno.+dos	la cadena "uno", seguida de uno o más caracteres, seguidos de la cadena "dos"

Los patrones del **grep** pueden entrar en conflicto con los metacaracteres del *shell* (los comodines, etc.), por lo que es necesario enmarcarlos entre comillas simples.

Por ejemplo, si tenemos un fichero de teléfonos donde aparecen los nombres y luego los números, y deseamos buscar las entradas correspondientes a personas en Madrid cuyo nombre empiece por "P", podríamos teclear

```
grep '^P.*91-' telefonos
```

Obsérvese el uso obligado de las comillas simples.

Algunas opciones interesantes del **grep**:

-v	imprime las líneas que no contienen el patrón especificado
-c	no imprime las líneas, sino el número de líneas que encajan
-l	imprime sólo los nombres de los ficheros conteniendo el patrón
-n	para cada línea, imprime además el número de línea dentro del fichero

Ejemplo:

```
grep -cv ";" *.c
```

nos da el número de líneas en los ficheros de extensión .c que no tienen un punto y coma.

16. Glosario de órdenes UNIX comunes

En estas líneas aparecen los nombres de algunas de las órdenes UNIX más utilizadas, clasificadas por el propósito que tienen, y acompañadas de un breve comentario sobre su función.

Proceso de textos

awk	lenguaje para buscar patrones, seleccionar campos, etc. en un fichero
banner	imprime un texto en caracteres grandes
cat	concatena, copia o imprime ficheros
cut	corta (extrae) campos de las líneas de un fichero
echo	imprime un texto
ed	editor de textos
emacs	editor de textos orientado a pantallas
expand	expande tabuladores a espacios, o viceversa
fold	Ajusta un texto a un número de columnas
grep	busca patrones en ficheros
head	imprime las primeras líneas de un fichero
join	Funde las líneas de dos ficheros que tienen campos comunes
more	imprime un fichero pantalla por pantalla
nl	imprime las líneas de un fichero con su número a la izquierda
paste	concatena líneas de ficheros distintos
pg	imprime un fichero pantalla por pantalla
printf	imprime un texto, aceptando múltiples formatos
rmnl	borra líneas en blanco superfluas
sed	"editor de flujos": edita no interactivamente líneas de ficheros
sort	ordena alfabéticamente las líneas de un fichero
tail	imprime las últimas líneas de un fichero
vi	editor de textos orientado a pantallas
wc	cuenta los caracteres, palabras o líneas de un fichero

Manipulación de ficheros

cd	cambia de directorio de trabajo
chacl	cambia los permisos de un fichero, con ACL
chmod	cambia los permisos de un fichero
chown	cambia el propietario de un fichero
cp	copia de ficheros y directorios
diff	compara ficheros
du	da estadísticas de espacio ocupado en disco
file	determina el tipo de fichero

find	encuentra ficheros
gunzip	descomprime ficheros
gzip	comprime ficheros
ln	crea enlaces en el sistema de ficheros
lsacl	lista los permisos completos de un fichero (ACL)
mkdir	creación de directorios
mv	renombrado o movimiento de ficheros y directorios
pack	comprime ficheros
pwd	imprime el directorio actual de trabajo
rm	borrado de ficheros
rmdir	borrado de directorios
script	registra en un fichero la sesión de terminal
split	trocea un fichero en fragmentos de tamaño fijo
tar	compacta ficheros en uno solo
tee	hace duplicados de la salida estándar
touch	cambia las fechas/horas de última modificación, etc.
umask	declara la máscara de permisos para cuando se crean nuevos ficheros
unpack	descomprime ficheros

Procesos y seguridad

chfn	cambia la información sobre un usuario
chsh	cambia el *shell* de un usuario
jobs	muestra las tareas de segundo plano
kill	envía una señal a procesos
nohup	ejecuta una orden inmune a salidas de sesión
newgrp	cambia de grupo de usuarios
passwd	cambia la clave de un usuario
ps	muestra los procesos activos
su	cambia a superusuario
time	ejecuta una orden y muestra los tiempos consumidos

Manejo de la terminal y sesión cotidiana

banner	imprime un texto en letras gordas
cal	imprime calendario
clear	borra la pantalla
date	imprime o altera fecha y hora
elm	utilidad de correo electrónico
exit	finaliza el *shell*

lock	deja bloqueada la terminal
lp	envía documentos a la impresora
mail	utilidad de correo electrónico
mailfrom	imprime los títulos de los mensajes de correo pendientes
man	ayuda en línea sobre órdenes y utilidades
mesg	bloquea/desbloquea la terminal ante mensajes
stty	muestra o altera los parámetros de la terminal
tty	imprime el nombre de la terminal
uptime	Lista compacta de usuarios activos y carga del sistema
who	muestra los usuarios activos
whoami	imprime el nombre del usuario
write	dialoga con otra terminal

Utilidades para escritos de *shell*

basename	extrae fragmentos de un nombre de fichero
exit	termina el *shell*
false	devuelve un uno
line	lee una línea
mktemp	genera un nombre de fichero temporal
read	lee una línea
sleep	hace una pausa
true	devuelve un cero
wait	espera la terminación de un proceso hijo
xargs	ejecuta una orden usando como argumentos la entrada estándar

17. Escritos del *shell* (*shell scripts*)

En esta sección presentaremos los llamados *shell scripts* o escritos del *shell*, que son ficheros de texto con órdenes para el *shell*. En un escrito de *shell* pueden aparecer construcciones más complejas que las estudiadas, como bucles, comentarios, etc., que constituyen todo un lenguaje de programación estructurada al estilo de Pascal o C.

17.1 Ficheros de texto ejecutables

Supongamos que existe una orden que usted repite con frecuencia. Podría ser el borrado recursivo de todos los ficheros .tmp desde su direcorio local, lo cual se puede hacer con

```
find $HOME -name "*.tmp" -exec rm {} \;
```

Pues bien, podemos crear un fichero cuyo contenido sea ese mismo texto. Si con **chmod** cambiamos sus permisos para que sea ejecutable, tendremos un nuevo programa disponible. Veamos un ejemplo:

```
$ cat >borra
find $HOME -name "*.tmp" -exec rm {} \;
^D

$ chmod +x borra

$ borra
```

El fichero **borra** ha pasado a ser un programa ejecutable, con lo que disponemos de un mecanismo para, por lo menos, acortar el nombre de las órdenes. Ésta es la base para construir los escritos de *shell*.

Escrito de *shell*: una secuencia de órdenes del *shell*.

Hagamos una matización sobre cómo se ejecutan estos programas: el procesador de órdenes, cuando le mandamos ejecutar un programa, intenta ejecutarlo como si fuera un fichero binario (es decir, escrito en código máquina). Si ve que no se trata de un fichero binario, lanza un *shell* hijo con el programa como entrada estándar.

En otras palabras, es como si un *shell* (hijo) leyera de un teclado imaginario el contenido del programa.

Los escritos de *shell* deberían comenzar por una línea consistente en:

```
#!/bin/ksh
```

(las últimas letras podrían ser **sh** o **csh**, dependiendo del shell que se desee utilizar). Esta línea o cabecera califica al fichero sin lugar a dudas como un ejecutable. En los ejemplos que se tratan aquí no hará falta incluir esta cabecera, pero puede hacerse necesaria para otras funciones.

17.2 Perfiles de usuario

El entorno de un usuario no nace de la nada. En el proceso de ingreso (*login*) al sistema, se cargan las variables de entorno, aparte de ocurrir otras acciones que, en suma, constituyen lo que se da en llamar **perfiles de usuario**.

Existen un par de ficheros encargados de especificar los perfiles de usuario, y varían según el procesador de órdenes de que se trate. Empezaremos con el **sh** y el **ksh**.

Cuando iniciamos una sesión con el **sh**, este procesador lee el fichero **/etc/profile**, el cual contiene valores iniciales de variables de entorno, mensajes de bienvenida y copyright, etc. Este fichero pertenece al superusuario y es común a todos los usuarios que entren con **sh**. Una vez tramitado el /etc/profile, se lee e interpreta el fichero **.profile** del directorio local del usuario. En este fichero cada usuario puede definir sus propios valores de entorno, ejecutar programas iniciales, etc.

Además, en el **ksh** puede hacerse que cada vez que se llama nuevamente al *shell*, se ejecute un escrito. Si el parámetro **ENV** está definido, su contenido será el nombre del fichero que se ejecuta. Lo habitual es hacer

```
ENV=$HOME/.kshrc
```

y en el fichero **.kshrc** escribir las órdenes oportunas (normalmente redefinir variables de entorno o alias).

La cuestión con el **csh** es muy similar, salvo que los ficheros tienen distintos nombres: se lee el **/etc/csh.login** a escala global, y el **.login** localmente para cada usuario. Además, se lee el fichero local **.cshrc** cada vez que se ejecuta el **csh**, incluida la primera.

Cuando se abandona una sesión iniciada con **csh**, se interpreta el fichero local **.logout**, que habitualmente se usa para mensajes de despedida o borrar ficheros temporales.

Eche un vistazo a los ficheros **.profile** y **.kshrc** de su directorio local, si existen.

17.3 Técnica de creación de los escritos

Antes de entrar en la sintaxis de los escritos, daremos unas guías para su creación. En principio, como son ficheros ordinarios de texto, se crearán con un editor o herramienta similar. En segundo lugar, es conveniente darles permiso de ejecución con **chmod**.

Por último, para ejecutar un escrito, se puede hacer de dos maneras:

a) si el fichero de escrito posee permiso de ejecución, escribiendo su nombre en la línea de órdenes.

b) si no es así, escribiendo **. fichero**, o en el **csh**, **source fichero**.

c) si no es ejecutable, escribiendo **sh fichero**.

La sintaxis de escritos que explicaremos será la del **ksh**, que es la misma del **sh** más algunos añadidos. La sintaxis del **csh** presenta algunas variaciones importantes respecto a la que vamos a describir, aunque es bastante similar.

17.4 Lecturas y escrituras

17.4.1 ECHO

La orden **echo** imprime un texto por la salida estándar.

Podemos añadir que ciertas combinaciones de caracteres, llamadas **secuencias de escape**, tienen significados especiales cuando van escritas en un texto entre comillas. Por ejemplo:

\a	señal acústica
\c	el "echo" no salta línea
\n	nueva línea (salto de línea)
\t	salto de tabulador
****	el carácter \

Ejemplo:

```
$ echo "un texto\n\ten dos líneas"
un texto
        en dos líneas
$
```

17.4.2 READ

Mediante la orden **read** podemos instruir a nuestros escritos para que pregunten al usuario:

read ***variable1 variable2*** ...

Las variables ***variable1***, ***variable2***, etc., adquieren los valores de las distintas palabras que escribamos en una línea. Diremos que una línea se divide en secuencias de caracteres (**palabras**) separadas por **caracteres separadores**, que por defecto son los espacios y tabuladores.

Ejemplo:

```
$ read var1 var2
una línea
$ echo "var1=$var1\nvar2=$var2"
var1=una
var2=línea
```

La última variable recoge todos los caracteres sobrantes de la línea:

© *José Miguel Santos Espino*

```
$ read var1 var2
una línea de cinco palabras
$ echo "var1=$var1\nvar2=$var2"
var1=una
var2=línea de cinco palabras
```

Una opción interesante permite presentar un texto previo a la lectura de la variable:

```
$ read x?"Valor de x="
Valor de x=<entrar aquí el valor de x>
```

17.5 Comentarios

Los escritos no dejan de ser programas, y como tales deberían admitir comentarios, o bien ilustrativos de qué hace cada fragmento del escrito, o bien para inactivar temporalmente alguna línea del escrito.

Las líneas que comienzan por el carácter almohadilla **#** se ignoran, así como todo lo que venga detrás de **#** en una línea cualquiera.

```
$ cat programa
# Esto es un comentario
echo Hola, mundo # feroz
# Otro comentario
$ programa
Hola, mundo
$
```

17.6 Parámetros a los escritos

Los escritos pueden aceptar parámetros y trabajar con ellos de una forma cómoda. Para ello se emplean unas variables predeterminadas de la forma **$c**, donde **c** es algún carácter. Veamos algunas de ellas:

$#	número de parámetros pasados
$*	la cadena formada por todos los párámetros
$0	primer parámetro
$1	segundo parámetro
...	
$9	noveno parámetro

Así, si construimos un escrito llamado **invierte** con este contenido:

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

podrá ocurrir algo como esto:

```
$ invierte uno dos tres cuatro
tres dos uno
$
```

Obsérvese que el parámetro **cuatro** no se visualiza, porque no está contemplado en el escrito.

17.7 Las órdenes devuelven un valor

Con el fin de escribir escritos complejos donde se ejecute un programa y se realicen acciones diferentes según haya terminado con éxito o no, el UNIX establece que todo programa, al terminar, devuelve un número entero. Para devolver ese valor se emplea la orden

exit ***valor***

Y el programa devolverá *valor* como "resultado". En ausencia de **exit**, se asume que el programa devuelve un cero. El cero se considera resultado de un programa correcto.

El valor de la última orden ejecutada puede consultarse con la variable **$?**

Ejemplo.

```
$ cat pepe.txt
hola, me llamo pepe.
$ grep hola pepe.txt
hola, me llamo pepe.
$ echo $?
0
$ grep nada pepe.txt
$ echo $?
1
```

Aquí, el programa **grep** devuelve un cero (correcto) en el primer caso, y un uno (error o fallo) en el segundo caso.

Un conducto devuelve el resultado del último programa que se ejecutó.

17.8 Evaluación de condiciones

Los escritos de *shell* tienen mecanismos para construir y evaluar expresiones aritméticas y lógicas. Una condición lógica o booleana es *cierta* o *falsa*, insistiendo en que "cierto" es un cero y "falso" un número no nulo.

Se admiten estas dos sintaxis para las expresiones booleanas:

[***expresión*** **]**
test ***expresión***

La primera opción, expresión entre corchetes, es la más usada y la que sugerimos.

Las expresiones, a su vez, se construyen de distinta forma según qué se quiera comprobar. Lo vemos a continuación.

17.8.1 Información sobre ficheros

Una sintaxis es

opción nombre

Donde *opción* es una opción que sirve para interpretar *nombre*, según estas posibilidades:

-e	el fichero existe
-f	es un fichero regular
-d	es un directorio
-r	este proceso tiene permiso de lectura sobre *nombre*
-w	este proceso tiene permiso de escritura sobre *nombre*
-x	es ejecutable
-s	tiene una longitud mayor que cero (no está vacío)

Si la condición es cierta, **test** devuelve un cero; si no, devuelve un uno.

Ejemplo:

[-f /usr/mail/pepe]

devuelve 0 si existe ese fichero (puede utilizarse para saber si el usuario "pepe" tiene correo).

Otras posibilidades con ficheros:

fich1 **-nt** *fich2* cierto si *fich1* es más reciente que *fich2*
fich1 **-ot** *fich2* cierto si *fich1* es anterior a *fich2*

17.8.2 Expresiones aritméticas

Este tipo de expresiones sirve para comparar números o cadenas de caracteres. La expresión puede ser un número, o una expresión de la forma **A OP B**, o bien **OP A**; donde **OP** puede ser uno de estos operadores:

-eq	igual
-ne	distinto
-gt	mayor que
-ge	mayor o igual
-lt	menor que
-le	menor o igual
=	igual (cadenas de caracteres)
!=	distinto (cadenas)
!	negación lógica
-z	cierto si es cadena vacía

Como en este ejemplo:

```
[ "$#" -lt 2   ]
```

devuelve un 0 si el número de parámetros es menor que 2.

17.9 Construcción condicional: if-then-elif-else

Por fin le daremos algún sentido a las expresiones condicionales. Podemos ejecutar condicionalmente partes de nuestro escrito con la sintaxis

```
if orden-condición
then
 órdenes
fi
```

Las *órdenes* sólo se ejecutarán si *orden-condición* devuelve un cero. Con un ejemplo:

```
if [ -f /usr/mail/$USER ]
then
 echo Tiene usted correo.
fi
```

Este escrito muestra un mensaje si cierto fichero está presente (en concreto, el fichero de correo del usuario que ejecuta el escrito).

Se aceptan alternativas con **elif** y **else**, como muestra el ejemplo:

```
if [ "$#" -eq 2 ]
then
 echo dos parámetros
elif [ "$#" -eq 3 ]
then
 echo tres parámetros
else
 echo uno o más de tres parámetros
fi
```

Se puede escribir una construcción if en una sola línea, separando sus componentes por punto y coma, como en este ejemplo:

```
if [ $LINEAS = 100 ]; then exit; fi
```

17.10 Secuenciación condicional: && y ||

El valor de retorno de una orden se puede utilizar para condicionar la ejecución de posteriores órdenes. En concreto, la combinación

```
orden1 && orden2 && orden3
```

se comporta según el siguiente algoritmo:

```
ejecuta orden1
si orden1 termina correctamente,
        ejecuta orden2
        si orden2 termina correctamente
                ejecuta orden3
si no, no hagas nada
```

Aquí, "termina correctamente" significa "retorna un valor igual a cero". El operador **&&** es bastante útil para evitar ejecutar secuencias inútiles de acciones si un paso previo falla. Como ejemplo:

```
[ -f listado_de_hoy ] && procesa listado_de_hoy
```

A veces interesa algo totalmente contrario: forzar la ejecución de una orden si una orden previa no va bien. Esto se realiza con el operador | |.

```
[ -f listado_de_hoy ] || echo hoy no hay listado
```

17.11 Construcción condicional case

La sentencia **case** permite decidir qué hacer, según lo que aparezca en una cadena. Su elaborada sintaxis es ésta:

case *cadena* **in**
patrón1 [**|** *patrón2* ...] **)** *lista de órdenes* **;;**
patrón3 [**|** *patrón4* ...] **)** *lista de órdenes* **;;**
...
esac

cadena es una cadena de caracteres (puede contener parámetros de *shell*), y se contrasta con los distintos patrones declarados en la sentencia. Si encaja con alguno de ellos, se ejecuta la *lista de órdenes* vinculada al primer patrón con el que encaja.

Los patrones son, o bien simples cadenas, o bien pueden contener caracteres comodín, como en la expansión de nombres de ficheros.

Ejemplo:

```
case $1 in
 *.txt ) echo "Una palabra *.txt" ;;
 a*    ) echo "Esta palabra empieza por \"a\"" ;;
 ?     ) echo "Esta palabra tiene una sola letra" ;;
 *     ) echo "Una palabra cualquiera" ;;
esac
```

La orden **case** es muy apropiada para hacer menús interactivos de opciones.

17.12 Bucle for

Esta construcción iterativa no es como la sentencia "for" de los lenguajes de programación convencionales, aunque también sirve para ejecutar repetidamente una sentencia, variando en cada paso el valor de una variable.

Su sintaxis es

for *nombre* **in** *lista*
do
    órdenes
done

nombre será un nombre de variable (que no tiene por qué existir previamente). En cada iteración, la variable *nombre* toma el valor del siguiente elemento de *lista*. Si se omite **in** *lista*, se ejecuta el bucle **for** una vez por cada parámetro pasado en la línea de órdenes.

Mejor es verlo con un ejemplo:

```
for param
do
 echo un parametro: $param
done
```

Si este escrito se llama **lista.param**, esta sería una posible ejecución:

```
$ lista.param uno dos tres
un parametro: uno
un parametro: dos
un parametro: tres
$
```

Para borrar todos los ficheros terminados en .c y .bak:

```
for fich in *.c *.bak
do
  echo "Borrando fichero $fich"
  rm $fich
done
```

Este ejemplo más sofisticado muestra recursivamente los ficheros a partir de nuestro directorio local que estén vacíos.

```
for f in $(find $HOME -print)
do
 if [ ! -s $f ]
 then echo "Fichero vacío: $f"
 fi
done
```

17.13 Construcciones while y until

Para ejecutar repetidamente una orden, disponemos de las construcciones

while *expresión* **do** *órdenes* **done**
until *expresión* **do** *órdenes* **done**

que repiten las *órdenes* mientras la *expresión* sea cierta (**while**), o mientras no lo sea (**until**). Se garantiza que el **until** se ejecuta al menos una vez.

Ejemplo:

```
x=1
while [ $x -ne 99 ]
do
  read x?"Entre un número: "
  echo "El número es $x"
done
echo "Terminé"
```

Lee cadenas y las imprime, hasta que se entre un 99.

Otro ejemplo es este programa que envía un mensaje cada cierto tiempo:

```
while true
do
 sleep 30
 echo Han pasado 30 segundos más
done
```

Este tipo de construcciones es muy empleado en los *demonios* del sistema.

17.14 Instrucciones break y continue

Para controlar el flujo de ejecución dentro de los bucles **for**, **while** y **until**, se dispone de dos instrucciones, **break** y **continue**, a imitación de las existentes en el lenguaje C.

17.14.1 BREAK

La instrucción **break** hace salir del bucle que la contiene.

Ejemplo: Busca la cadena "hola" en los ficheros de extensión .txt; en cuanto encuentre uno termina.

```
for f in *.txt
do
  grep hola $f
  if [ $? ]
  then echo "Encontré un fichero"; break
  fi
done
```

Si **break** se encuentra dentro de dos bucles anidados, se sale del más interno.

17.14.2 CONTINUE

La instrucción **continue** se salta el resto de las instrucciones hasta el fin del bucle correspondiente, y empieza inmediatamente otra iteración.

Ejemplo: no tratar los ficheros ejecutables

```
for fich
do
  if [ -x $fich ]
  then continue
  fi
  ... ejecutar lo que sea con $fich ...
done
```

17.15 Expresiones aritméticas: let

El procesador de órdenes **ksh** permite trabajar con expresiones numéricas de una forma cómoda mediante la construcción **let**, una de cuyas sintaxis puede ser

let ***variable*** **=** ***expresión***

donde *expresión* es una expresión aritmética que puede incluir nombres de variables, operadores matemáticos y paréntesis. La precedencia es la habitual (producto y división antes que suma, etc.) Eso sí, la expresión no debe contener espacios separadores.

Ejemplo: el típico bucle contador.

```
x=1
while [ $x != 10 ]
do
  echo $x
  let x=x+1
done
```

17.16 Las construcciones también son órdenes

Como dice el título del apartado, una construcción **for**, **while**, **case**, **if**, o cualquier otra de las que hemos presentado es a todos los efectos una orden del *shell*. Esto implica que les podemos aplicar redirección, las podemos insertar en conductos (*pipes*), las podemos dejar en segundo plano, y en general les podemos aplicar cuantas técnicas y virguerías se hayan descrito para ejecutar órdenes en UNIX.

Para intentar aclarar un poco esta cuestión, vaya una serie de ejemplos:

```
for f in uno dos tres cuatro
do
  echo "Este es el número $f"
done >salida

for f in $(ls /usr)
do
  echo $(ls -s $f) " bytes"
done | sort -n | more

cat *.txt | while true
do
  read x
  if [ $x = fin ] break
  echo $x
done
```

17.17 Funciones

El lenguaje de los *shells* de UNIX incorpora cierta modularidad al permitir la definición de **funciones**. Una función podría verse como una orden que se define dentro del propio escrito de *shell* y que se puede invocar dentro de él.

Hay dos sintaxis para definición de funciones (sin diferencia de significado):

function ***nombre*** **{** ***cualquier escrito de shell*** **}**
nombre **() {** ***cualquier escrito de shell*** **}**

Para invocar a una función, simplemente se escribe su nombre, seguido de los argumentos que se le quieran pasar.

En el cuerpo de la función, se pueden utilizar los argumentos suministrados de igual forma que hacemos en los escritos de *shell*: **$1**, **$2**, etc.

Ejemplo:

```
function lista
do
  ls -l $1
done

lista pepe.txt
```

Una función, como cualquier otra orden, devuelve un valor, que por omisión es el de la última orden que se ejecutó antes de salir de la función.

Las funciones pueden devolver un valor concreto con la instrucción **return**. Si se ejecuta dentro de una función

return *n*

se abandona la función, dando como valor *n*.