El Sistema Operativo Unix

Fernando Magariños Lamas

3 de junio de 1999

Contenido

Índice de Tablas

Objetivos

- Identificará el concepto de Sistema Operativo
- Identificará los elementos del S.O. Unix
- La estructura de archivos
- Comandos de navegación
- El uso de un editor

Capítulo 1

Antecedentes

Entendemos por sistema operativo al programa que sirve de enlace entre el usuario, las aplicaciones y el sistema de archivos, así como entre las aplicaciones y los dispositivos y entre las propias aplicaciones en el caso de sistemas operativos que permiten varias tareas simultáneamente.

En una extensión de la idea anterior están los sistemas operativos multi usuarios, donde además el sistema operativo puede proveer la comunicación entre aplicaciones de usuarios, como en el caso de Unix[1].

El propósito primordial de un sistema operativo es el de servir de soporte a los programas que el usuario utiliza. Por ejemplo, el editor que se utiliza para elaborar un documento. Este editor no podría correr por sí mismo sin el soporte que el sistema operativo le otorga, como es el caso de recibir e interpretar los carácteres que el usuario pulsa en el teclado y mostrarlos en la pantalla, salvarlos en el disco bajo una cierta estructura que llamaremos sistema de archivos, e incluso controlar el puerto por donde se transmiten esos mismos carácteres a la impresora.

Por otra parte, todos los sistemas operativos incluyen una serie de pequeños programas o utilerías que o bien colaboran con el sistema operativo complementando las tareas, o son utilizados por los usuarios para hacer más eficiente su trabajo.

Los sistemas operativos pueden ser minimalístas (como es el caso de MS-DOS) o grandes y complejos (como OS/2 y VMS). Unix se sitúa en algún punto intermedio de estas dos categorías y es cada vez más difícil

[1] En adelante toda referencia a UNIX será al sistema operativo con marca registrada por X/Open y Unix a versiones derivadas de éste

discernir su situación exacta. Aunque provee de más recursos y hace más que los primeros sistemas operativos, no alcanza a los sistemas operativos más avanzados. Ejemplos de éstos últimos son incluso antecedentes de Unix, como CTSS y Multics, de los cuales hablaremos más adelante.

El desarrollo de los sistemas operativos va necesariamente ligado al desarrollo del soporte físico, comúnmente llamado hardware. Es así que las primeras computadoras no tuvieron sistema operativo, dado lo primitivo de sus medios de almacenamiento que dependían de la alimentación eléctrica directa.

El primer sistema operativo fue desarrollado en el Centro de Investigación y Desarrollo de la General Motors Company a principio de la década de los cincuenta. Sólo contemplaba la ejecución de una tarea a la vez en un método llamado de bloques o *batch*.

En la siguiente década comenzaron a aparecer los sistemas operativos multi tareas, que eran capaces de procesar varios trabajos en un tiempo dado a base de repartir los tiempos de proceso. Surgieron a partir de la necesidad de aprovechar al máximo el uso del procesador. Los ingenieros de desarrollo habían observado que buena parte del tiempo empleado en ejecutar un programa, transcurría en ciclos donde el procesador central esperaba a que terminase una acción de lectura o escritura a uno de los dispositivos. Estos tiempos muertos podían ser aprovechados si momentáneamente se le pasaba al procesador otra tarea a ejecutar.

Hasta este punto la visión de uso era la siguiente: las tareas se le asignan al procesador y permanecen con el control de la computadora hasta que son terminadas. Desde el momento que se visualiza el primer sistema operativo multi tarea, el cambio es radical: una aplicación (el sistema operativo), será quien decida como se aprovecha al procesador, de tal manera que conceptualmente es ahora el procesador el que destina el orden en que las tareas serán ejecutadas.

Veamos esto con un ejemplo: en un restaurante tenemos a un sólo cocinero quién va preparando los platillos conforme le van siendo entregados. Cada nueva orden traida por alguno de los meseros se coloca debajo de las anteriores y el cocinero únicamente toma la que está arriba. Obviamente tiene tiempos muertos en los que espera a que, por ejemplo, un faisán se cocine. Este método es bastante torpe, dado que preparar un faisán puede tomar 5 minutos en sazonarse y 30 en cocción más otros 2 en lo que se presenta en el plato. Los 30 minutos de cocción pudieron ser aprovechados para tomar la siguiente orden, comenzar a prepararla y repetir el proceso hasta que se tenga ocupado todo el tiempo del cocinero.

La siguiente generación consolida la optimización del uso de recursos y se introduce el concepto de *timesharing* que consiste en el acceso en tiempo real de varios usuarios simultáneos. Esto permitió el desarrollo sostenido de aplicaciones cada vez más complejas y en particular de mejores sistemas operativos.

Por otra parte, el desarrollo de sistemas se ve complementado con la intrusión de computadoras en las universidades donde se pone mayor énfasis en la interacción con el usuario. Este último factor se ve beneficiado también por las repetidas caidas en los precios de equipos cada vez más sofisticados.

Con los sistemas operativos que precedieron éste periodo, el tiempo de desarollo de aplicaciones era demasiado grande. Anteriormente, el ciclo que comprendía la edición, compilación y corrección del programa podía tomar incluso días, ya que con frecuencia la computadora se encontraba a gran distancia del usuario. Las tareas eran perforadas en tarjetas, leidas por la computadora y en este paso podía ocurrir que la tarea no pudiese ser llevada a cabo por culpa de una coma fuera de lugar, produciendo un listado con los errores que, probablemente, el usuario no podría ver sino hasta unos dias después.

Al simple hecho de poder tener una respuesta inmediata de la computadora, podemos agradecer gran parte de los adelantos que hemos podido disfrutar en los últimos 30 años.

Mencionamos antes a CTSS y a Multics como precursores de Unix. Ambos fueron desarrollados en el MIT[2], como parte de programas de investigación y desarrollo de sistemas operativos *timesharing*, y significaron un gran avance en términos de llevar al extremo las ideas de optimización de recursos computacionales. Ambos fueron escritos en lenguajes de alto nivel, que sería la idea seminal de Unix. La eficacia de esta manera de trabajo quedó demostrada cuando Multics fue escrito utilizando al mismo CTSS como soporte.

En 1965, los Laboratorios de Telefonía Bell (una división de AT&T) trabajaban en conjunto con General Electric y el Proyecto MAC del MIT en el desarrollo de Multics. Por diversas razones, Bell se separó del grupo pero al seguir con la necesidad de un sistema operativo con características

[2]Massachusets Institute of Technology

similares, Ken Thompson y Dennis Ritchie decidieron diseñar un sistema operativo que llenara los requisitos predispuestos. En 1970, Thompson lo implementó como un ambiente de desarrollo en una PDP-7. A modo de mofa hacia Multics, Brian Kernighan lo llamó UNIX.

Algún tiempo después Ritchie desarrolló el lenguaje C de programación. En 1973 UNIX fué reescrito en C, lo cual, como veremos más adelante, lo impulsó a los niveles de popularidad en que se encuentra hoy. En 1977 UNIX fué portado a otra máquina con una arquitectura diferente a la de PDP, gracias a que, una vez transportado el compilador de C, fue recompilado con tan sólo los cambios necesarios para adecuarlo a la nueva arquitectura. Aquí nace el concepto de sistemas abiertos al que se le da tanta importancia ahora.

Al momento de diseñarlo se le dió preponderancia a un concepto revolucionario: estaría compuesto de pequeños programas de gran generalidad de tal manera que se pudiesen interconectar para realizar tareas mayores.

Su gran portabilidad facilitó que se implementara en computadoras disímiles en universidades y centros de investigación, y que posteriormente fuese utilizado en aplicaciones comerciales.

Existen dos corrientes principales: System V de Unix System Laboratories[3] y BSD (Berkeley Software Distribution) de la Universidad de Berkeley, California. La versión de USL se encuentra en su cuarta revisión, o SVR4, mientras que la última versión que se hizo de BSD fué la 4.4 debido a diversos problemas que terminaron con una demanda de USL hacia BSD y una contra demanda de BSD a USL y el posterior abandono del grupo de Berkeley al considerar a su Unix un producto terminado y ya no de investigación.

Todos los derivados más populares toman lo mejor de cada uno de ellos. En adelante haremos referencia a particularidades tanto de SVR4 como de BSD por sus siglas.

[3] Unix System Laboratories (USL) originalmente de AT&T, fué vendida hace poco a Novell

Capítulo 2

Introducción

En este capítulo se tratarán algunos conceptos importantes de Unix para familiarizarnos con él.

Como hemos mencionado, Unix es un sistema operativo multi usuario y multi tarea, lo cual quiere decir que pese a que lo utilicemos en una computadora personal, puede tener varios usuarios simultáneos conectados por una *terminal.* Una terminal puede ser nada más un monitor y un teclado u otra microcomputadora. El hecho es que, a diferencia de una PC o una Macintosh, un sistema Unix no está dedicado exclusivamente a nuestra atención, sino a la de varios usuarios. Aún en el caso de que seamos los únicos usuarios en el sistema, no podemos disponer de la máquina como en el caso de las microcomputadoras personales: el sistema operativo tiene sus propias tareas corriendo, como un usuario más, y por esto se debe apagar la máquina con cuidado de avisarle al sistema operativo que detenga sus procesos. De otra manera correremos peligro de perder información.

Debemos, desde el principio, comprender que Unix es un sistema operativo que fué diseñado con la idea de facilitar la vida de los programadores. Al usuario común, en ocasiones le parecerá demasiado complejo y hostíl. Pero debemos tener en cuenta que cuánto más lo conozcamos, mejor provecho podremos tener de él. No sólo es para usarlo, sino para programar en él.

En algunas ocasiones, en particular con los Unix derivados de SVR4, tendremos problemas en el sentido de que se puede *congelar* la terminal o que perdamos respuesta de la máquina o que simplemente se desconfigure la terminal. Esto de ninguna manera quiere decir que el sistema se haya caido. Sólo que hay problemas con la terminal. En casos cómo éste, lo peor que se puede hacer es apagar y prender la máquina de nuevo, como en el caso de las microcomputadoras. Lo mejor es avisar al encargado del sistema o entrar desde otra terminal.

2.1 Convenciones tipográficas

Se utilizarán *itálicas* cuando se introduzcan nuevos términos, se incluyan palabras de otros idiomas o se haga referencia a un comando, instrucción o programa.

En los ejemplos, el texto aparecerá siempre en `tipo máquina de escribir`.

Se reserva el uso de negritas para títulos.

2.2 Identificación del teclado

Habrá algunos carácteres que tienen significado especial en Unix. En su momento los iremos introduciendo junto con su definición. De momento basta con identificar unos cuantos para unificar el lenguaje que utilizaremos. Es muy importante diferenciar entre las dos diagonales, la tradicional `/` y la que llamaremos diagonal invertida `\`. Al carácter `#` lo llamaremos hash, por su nombre en inglés. Al carácter `@` lo llamaremos arroba, al `%` porciento, al `$` pesos, al `|` simplemente pipe[1], al `&` ampersan y al `^` circumflejo.

2.3 Entrar en sesión

Para poder utilizar un sistema Unix, es necesario tener una *cuenta* de acceso. Esta cuenta estará dada por el administrador del sistema. Típicamente en un sistema con pocos usuarios, el nombre de la cuenta coincidirá con el nombre del usuario y en un sistema grande será el apellido o la composición de nombre y apellido, o simplemente un alias. Digamos que tenemos al usuario Pablo Flores y es el único que lleva el nombre de Pablo en nuestro trabajo, entonces lo más probable es que su cuenta sea simplemente `pablo`.

[1]Se pronuncia como paip.

De otra manera, podría ser `pflores` o `pf17ss` o cualquier combinación que lo identifique en forma única a lo largo del sistema.

Después de iniciar el sistema operativo, que típicamente será tarea del administrador del sistema, en las terminales aparecerá un mensaje similar al siguiente:

```
Welcome to Linux 1.2.8.

ZootAllures login:
```

donde, obviamente, el nombre del sistema operativo y la versión así como el nombre de la máquina, serán diferentes, lo único que podemos garantizar que será igual es la palabra `login:`. Esto nos indica que el sistema está listo para aceptar la entrada de un usuario. Para esto es necesario identificarnos dando el nombre de nuestra cuenta y a continuación aparecerá:

```
Welcome to Linux 1.2.8.

ZootAllures login: pablo
Password:
```

En este punto Pablo deberá escribir su *password* o palabra secreta. Cuando la cuenta es creada, el password es fijado por el administrador del sistema. Es conveniente que la primera vez que se entra en sesión[2] se cambie el password. Esto se hace utilizando el comando `passwd` que primero nos pregunta el password anterior y después deberemos de dar el nuevo en dos ocasiones para garantizar que fue bien escrito:

```
$ passwd
Changing password for pablo
Enter old password:
Illegal password, imposter.
$ passwd
Changing password for mancha
Enter old password:
```

[2]Acción que comprende dar cuenta y password, y que en adelante llamaremos entrar en sesión.

```
Enter new password:
Re-type new password:
The password must have both upper- and lowercase
letters, or non-letters; try again.
Enter new password:
Re-type new password:
Password changed.
```

Como podemos observar, la primera vez no se dió correctamente el password con lo cual el sistema no acepta la operación de cambiarlo. En la segunda ocasión, después de escribir el nuevo password dos veces, nos indica que es conveniente que usemos un password con carácteres en mayúsculas y minúsculas, así como dígitos y signos de puntuación.

No hay manera de garantizar un password completamente seguro, pero es buena costumbre no usar palabras que tengan sentido ni nombres propios. Si la hija de Pablo nació en 1989 y se llama Andolza, Andolza1989 sería un pésimo password para Pablo, ya que varios de sus colegas conocen el hecho además de que sería fácil de averiguar.

Sería igual de malo un password como Emma.Sunz si sus colegas saben que Pablo es un lector apasionado de Borges.

Un método tan bueno como cualquier otro para escoger un password, es tomar una frase, no muy conocida, utilizar, por ejemplo, el segundo o tercer carácter de cada palabra que la forma, intercalarle dígitos y signos de puntuación y poner algunas de las letras en mayúsculas. Por ejemplo, con el verso del poeta Maiakovsky "No es difícil morir en esta vida, que vivir es más fácil" tomamos las palabras de más de cinco letras: `difícil morir fácil` y de cada una tomamos la cuenta del número de letras y los carácteres 3 y 5: 7fc5rr5cl. Luego de haber hecho esto, y además publicarlo, éste método queda totalmente invalidado. Es mejor que cada persona idee su propio método para seleccionar el password —que además es conveniente cambiar con cierta frecuencia—, y que le sea fácil de recordar en todo momento. Nunca se debe de escribir un password en ningún sitio. Mucho menos tatuarselo en el pecho, que es de mal gusto y además como se habrá de cambiar con frecuencia, termina uno con la epidermis hecha un asco de tanto *password* tachado.

Habrá sistemas donde en lugar de con una aburrida pantalla de texto nos encontraremos con una excitante ventana en modo gráfico, pero el procedimiento será siempre el mismo.

Capítulo 3

Sistema de Archivos

Entederemos por *archivo* a un conjunto de carácteres o *bytes*. El sistema de archivos no impone ninguna estructura sobre de el archivo y su contenido no tiene ningún significado para el sistema de archivos, el significado depende exclusivamente de los programas que lo manipulen.

La estructura del sistema de archivos es jerárquica, es decir una gráfica dirigida o, vista de otro modo, una estructura arbórea.

El directorio principal, llamado raíz, tiene por nombre simplemente '`/`' que a su vez es el carácter utilizado para separar los nombres de los subsguientes directorios. Por ejemplo el directorio de trabajo de un usuario determinado puede ser: `/home/pablo` y el archivo donde se encuentran los comandos de arranque en el Unix de BSD está en: `/etc/rc.local`.

En SVR4 los nombres de los archivos generalmente tienen máximo 14 carácteres mientras que en BSD el tope son 255.

Al ser Unix un sistema operativo multi usuario, tiene también un modo de protección al sistema de archivos, previsto para la privacidad entre usuarios. Esta formado por tres tipos de accesos: de usuario, de grupo y de "otros". Es decir, el usuario fija los accesos para él, para las personas en su grupo y para el resto de los usuarios. En cada caso se tiene un permiso independiente para lectura, escritura y ejecución.

Varios usuarios pueden pertenecer al mismo grupo y un usuario puede, a su vez, pertenecer a varios grupos. Es decir, todos los usuarios del departamento de contabilidad pueden estar en el grupo `conta`[1], los de ingeniería en

[1]En realidad no hay ninguna razón para utilizar abreviaturas.

el grupo `inges` y a su vez, los jefes de ambos departamentos pueden estar en el grupo `jefes` además de en los antes mencionados.

El fin de esta caracterización de los usuarios es para permitir y restringir accesos a determinadas partes del sistema. En el ejemplo anterior, es obvio que ninguno de los usuarios, aparte de los del departamento de contabilidad, pueden tener acceso a los archivos donde se encuentra la nómina. A su vez, habrá directorios donde sólo los jefes de departamento podrán tener acceso y que nadie más (salvo el gerente y el administrador de la máquina) podrán leer y modificar los archivos contenidos.

3.1 Manejo del sistema de archivos

El comando utilizado para visualizar el contenido de un directorio es `ls` con una serie de opciones. Las opciones a programas y comandos en Unix típicamente van precedidas por el carácter `-`, salvo cuando se indique lo contrario. Las opciones sirven generalmente para cambiar el comportamiento de un comando. Por ejemplo para ver el contenido de un directorio en forma somera, basta dar `ls`:

```
$ ls
TODO                    funny.tex       ps-files
bib.tex                 gpl.tex         shell.tex
commands.tex            guide.dvi       shell2.tex
config.tex              guide.ps        starting.tex
convntns.tex            guide.tex       thanks.tex
copyright.tex           intro.tex       unixinfo.tex
emacs-chapter-stuff     kmsg            vi.tex
emacs.tex               kmsg.2          x11.tex
errors.tex              l               gpl.tex
find_unfinished         network.tex
```

o en forma mas completa `ls -al`, donde la opción `a` indica que queremos todos los archivos (se puede "ocultar" un archivo a `ls`) y la opción `l` indica que queremos la versión completa o "larga" de la salida:

```
$ ls -al
total 2431
drwxr-xr-x      4       mancha  users   1024    Sep     20      16:12   .
drwxrwxrwx      3       mancha  users   1024    Jan     8       19:52   ..
-rw-r--r--      1       mancha  users   572     Aug     7       22:08   TODO
-rw-r--r--      1       mancha  users   619     Feb     9       1994    bib.tex
-rw-r--r--      1       mancha  users   20051   Aug     3       21:10   commands.tex
-rw-r--r--      1       mancha  users   46025   Aug     30      19:32   config.tex
-rw-r--r--      1       mancha  users   1707    Sep     6       22:50   convntns.tex
-rw-r--r--      1       mancha  users   1945    Aug     30      16:57   copyright.tex
drwxr-xr-x      2       mancha  users   1024    Aug     5       20:25   emacs-chapter-stuff
-rw-r--r--      1       mancha  users   58341   Sep     6       22:51   emacs.tex
-rw-r--r--      1       mancha  users   7260    Aug     3       10:53   errors.tex
-rwxr-xr-x      1       mancha  users   20      Feb     9       1994    find_unfinished
-rw-r--r--      1       mancha  users   28447   Aug     30      16:51   funny.tex
-rw-r--r--      1       mancha  users   18124   Feb     9       1994    gpl.tex
-rw-r--r--      1       mancha  users   498708  Sep     20      16:11   guide.dvi
-rw-r--r--      1       mancha  users   1576525 Sep     20      16:11   guide.ps
-rw-r--r--      1       mancha  users   4721    Sep     8       13:35   guide.tex
-rw-r--r--      1       mancha  users   12714   Aug     3       10:30   intro.tex
-rw-r--r--      1       mancha  users   1177    Feb     9       1994    kmsg
-rw-r--r--      1       mancha  users   91      Feb     9       1994    kmsg.2
-rw-r--r--      1       mancha  users   25582   Aug     4       12:15   lgpl.tex
-rw-r--r--      1       mancha  users   5052    Aug     30      16:36   network.tex
drwxr-xr-x      2       mancha  users   1024    Aug     3       21:11   ps-files
-rw-r--r--      1       mancha  users   29551   Aug     4       12:01   shell.tex
-rw-r--r--      1       mancha  users   34042   Aug     4       13:09   shell2.tex
-rw-r--r--      1       mancha  users   22141   Aug     4       12:01   starting.tex
-rw-r--r--      1       mancha  users   934     Aug     2       20:45   thanks.tex
-rw-r--r--      1       mancha  users   10919   Aug     30      16:58   unixinfo.tex
-rw-r--r--      1       mancha  users   32095   Aug     30      19:35   vi.tex
-rw-r--r--      1       mancha  users   13574   Aug     30      18:24   x11.tex
```

Las primeras diez columnas indican el tipo de acceso que tiene cada archivo para los diferentes usuarios. Así, el archivo con nombre `guide.ps` tiene los accesos que se indican en la tabla 3.1

En el caso de los archivos ejecutables, aparece una `x` para denotarlos. Los caracteres para los archivos "especiales" son: `-` normales, `d` directorios, `l` liga simbólica, `b` dispositivo por bloques, `c` dispositivo por carácteres y `p` *pipe*[2] "nombrado".

Para cambiar los atributos de un archivo contamos con la instrucción `chmod`, que nos permite cambiar el modo de acceso del archivo. La sintáxis es:

```
chmod quienes operacion modo
```

[2]Como veremos más adelante, la sintáxis del sistema de archivos de Unix permite crear canales de comunicación entre procesos, como si éstos se tratasen de archivos.

```
         1    1        1      1        1    1    1      1      1
1234567890    1        2      3        4    5    6      7      8
-rw-r--r--    1     pablo   users   1576525 Sep   20     16:11  guide.ps
```

Tabla 3.1: Accesos a un archivo. Cada columna está numerada, con los siguientes significados: 1) No es un archivo "especial" 2) El dueño (pablo) puede leerlo 3) El dueño puede escribirlo 4) El dueño no puede ejecutarlo 5) Los del grupo (users) pueden leerlo 6) Los del grupo no pueden escribirlo 7) Los del grupo no pueden ejecutarlo 8) Los otros pueden leerlo 9) Los otros no pueden escribirlo 10) Los otros no pueden ejecutarlo 11) Número de ligas 12) Usuario 13) Grupo 14) Tamaño en bytes 15) y 16) Fecha de alteración 17) Hora de alteración 18) Nombre del archivo.

donde `quienes` puede ser una combinación de `ugo` con `u` para fijar el modo para el usuario, `g` para el grupo y `o` para los otros, es decir, para los que no son el usuario y que tampoco pertenecen al grupo. La `operacion` es `+` para añadir y `-` para retirar el modo a los grupos seleccionados y `modo` puede ser cualquier combinación de `rwx` con el significado dado en la tabla 3.1. Por ejemplo: si el usuario Pablo quiere que un archivo en particular, digamos que se llama `diario`, no sea leído por nadie en el sistema salvo por él y por *root*[3], tendría que usar: `chmod go-rw diario`. Así que si ahora examina los accesos del archivo con `ls`, estos serán:

```
-rw-------   1 pablo    usuarios      1789 Jun 13 01:54 diario
```

Lo mismo en caso de querer ocultar un directorio, por ejemplo el directorio `cursilerias`: `chmod go-rwx cursilerias` de tal manera que los demás usuarios del sistema ni siquiera podrán ver el contenido del directorio.

Ahora pensemos que Pablo es un gran programador[4] y en sus ratos libres se dedica, además de escribir cartas, a escribir programas que desea com-

[3] *root* es el usuario con todos los privilegios en el sistema. También es llamado el *superusuario*, que es típicamente el administrador y además no se le puede esconder nada.

[4] En la mitología Unix se les conoce como *hackers*, pese a que el término ultimamente a tendido a ser confundido con el de *crackers*, que se refiere a los *hackers* que se dedican a violar la integridad de los sistemas, pero cabe aclarar que no todos los *hackers* son *crackers* ni todos los *crackers* son verdaderos *hackers*.

partir con los demás usuarios del sistema. Para esto, crea un subdirectorio donde los depositará y además los dejará con los permisos adecuados para que todo mundo los pueda ejecutar y examinar. Digamos que decide que el directorio se llamará `bin`[5], y allí deposíta todos los programas que hasta ahora ha hecho. La secuencia completa es la siguiente:

```
1:$ cd
2:$ mkdir bin
3:$ chmod u=rwx bin
4:$ chmod go=rx bin
5:$ cd bin
6:$ cp ../fuentes/* .
7:$ chmod u=rwx *
8:$ chmod go=rx *
```

Ejercicio: ¿Puede explicar paso a paso lo que hizo Pablo?

Instructor: En la línea 1, se cambia a su directorio home, en la línea 6 copia todos los archivos de su directorio fuentes al nuevo directorio bin y a continuación fija los permisos adecuados.

La primera línea desplegada por ls es el total de bloques ocupados por el directorio:

```
total 2431
```

Las siguientes dos líneas se refieren a dos directorios especiales: el directorio '.' que es una referencia a sí mismo y el directorio '..' que es la referencia al directorio padre del actual.

```
drwxr-xr-x     4     mancha  users   1024    Sep     20      16:12    .
drwxrwxrwx     3     mancha  users   1024    Jan     8       19:52    ..
```

Encontramos también un directorio:

```
drwxr-xr-x   2 mancha   users        1024 Aug  5 20:25 emacs-chapter-stuff
```

donde el usuario, los miembros de su grupo y los demás usuarios tienen puesta la bandera de ejecución. En el caso de un directorio, esta bandera indica que se pueden cambiar a ese directorio, pero sólo el usuario tiene permiso de escritura.

[5]Este nombre de ninguna manera es fortuito, en Unix los programas que ejecutan los usuarios están en los directorios `/bin` y `/usr/bin`, donde *bin* es el apócope de binarios.

El archivo `find_unfinished` aparece también con la bandera de ejecución puesta. Como es un archivo ordinario, podemos asumir que realmente se trata de un "ejecutable", pese a medir tan sólo 20 bytes. Más adelante veremos cómo es posible esto.

```
-rwxr-xr-x    1    mancha  users   20    Feb    9     1994 find_unfinished
```

El comando `ls` es de los que más opciones tiene. Conocerlas todas es demasiado pedir pese a que probablemente sea de los comandos más usados. ¿Qué hacer entonces? Bueno, el segundo comando más empleado en Unix es `man`, que es el comando que muestra la ayuda de otros comandos.

Dadas las diferentes versiones de Unix que existen es difícil hacer un compendio completo de todas las opciones de cada comando, así que `man` debe ser siempre nuestra referencia para cada sistema. En este caso, para conocer bien todas las opciones de `ls`, lo más conveniente es que las veamos en la "página" de `ls`, por supuesto con `man`.

Las opciones más útiles son `-k` que lista las páginas de manual de los comandos referentes a la siguiente palabra que se dá, por ejemplo:

```
$ man -k change
chdir (2)           - Change working directory
chmod (1)           - Change the access permissions of files
chown (1)           - Change the user and group ownership of files
passwd (1)          - Change password
```

lista todos los comandos que contienen la palabra *change*. La otra opción más usada, sirve como una referencia rápida para saber que acción ejecuta un comando en particular. Es la opción `-f`. Por ejemplo:

```
$ man -f chdir
chdir (2)           - Change working directory
```

nos indica que `chdir` es el comando para movernos entre directorios.

Ahora ya sabemos como interpretar la salida de `ls`, como invocar la ayuda de un comando en particular, como buscar ayuda acerca de un tópico y como averiguar que hace un comando en particular. Estamos listos para comernos el mundo de Unix. O casi.

Para finalizar ésta sección hablaremos de algunos comandos para manipular archivos y directorios.

En caso de que quisiéramos ver el contenido de un archivo podríamos usar el comando `cat`:

```
$ cat find_unfinished
grep "\*\*\*" *.tex
```

Bien ahora conocemos el contenido de ese misterioso archivo ejecutable. Parece ser que fue escrito por algún niño o por una persona que habla otro idioma. `grep` y una serie de garabatos... Pero tenemos a `man` para saber si `grep` se trata de algún comando:

```
$ man -f grep
grep, egrep, fgrep (1)  - Print lines matching a pattern
```

parece ser que sí, que se trata de un comando, y uno bastante interesante ya que habla de imprimir las líneas que cumplen con un patrón[6] dado. Más adelante hablaremos de él, ahora listemos el contenido de otro archivo. Como lo más probable es que no se tenga a la mano un directorio como el que hemos estado mostrando en los ejemplos, vamos a usar uno común a todos los Unix, el directorio `/etc` y el archivo `passwd`, que es donde el Sistema Operativo coteja a los usuarios válidos en el sistema:

```
$ cat /etc/passwd
root:qb7Y7hJrPw69s:0:0:root:/root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:
lp:*:4:7:lp:/var/spool/lpd:
sync:*:5:0:sync:/sbin:/bin/sync
shutdown:*:6:0:shutdown:/sbin:/sbin/shutdown
halt:*:7:0:halt:/sbin:/sbin/halt
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/usr/lib/news:
uucp:*:10:14:uucp:/var/spool/uucppublic:
operator:*:11:0:operator:/root:/bin/bash
games:*:12:100:games:/usr/games:
man:*:13:15:man:/usr/man:
postmaster:*:14:12:postmaster:/var/spool/mail:/bin/bash
ftp:*:404:1::/home/ftp:/bin/bash
```

[6]Patrón en un sentido muy amplio. Podría decirse también modelo, descripción general o el término que utilizaremos más adelante: *expresión regular*, en la sección 5.4.

```
gonzo::418:100::/home/gonzo:/bin/bash
satan::419:100::/home/hell:/bin/bash
snake::420:100::/home/pit:/bin/bash
mancha:Ncoo.YlGCT1sU:501:100:La Mancha de la Calabaza que ladra:/home/mancha:/bin/bash
cecilia:PbcENyydKMFQo:502:100:Cecilia Estrada de Pavia:/home/cecilia:/bin/bash
petra:EnRaT.5l3w33s:503:100:Petra de la Parra Saucedo:/home/petra:/bin/bash
ruperto:JoNBYAhSOCenM:504:100:Ruperto Felgu\'erez P\'\i{}rez:/home/ruperto:/bin/bash
bill:AcU56dAK5xVKE:505:100:William Clinton:/home/bill:/bin/bash
cuau:OiuBXuoDFQOrc:506:100:Cuauht\'emoc C\'ardenas Sol\'orzano:/home/cuau:/bin/bash
```

Pues resultó que este archivo tiene más líneas de las que caben en la pantalla. Para poder visualizarlo pantalla por pantalla podemos usar un paginador, `more`, con el comando `cat /etc/passwd | more`. Si ejecutamos este comando veremos que tenemos manera de detener la salida hasta que demos un espacio y cambia de pantalla, ¿pero qué fué eso de `| more`? Bueno, pues simplemente le estamos pidiendo al sistema operativo que la salida producida por el comando `cat` se la pase como entrada al comando `more`. Esta técnica se conoce como *piping* o “entubado”. En la introducción hablábamos de que el diseño de Unix tenía en mente el reunir una colección de pequeños programas muy generales que nos permitan unirlos o conectarlos para realizar tareas más complejas. Este es el primer ejemplo al respecto, más adelante veremos construcciones complejas e interesantes.

Claro que si el usuario avezado lee la página del manual de `more`, podrá ver que no hace falta pasar por `cat` y el `pipe`, sino que directamente podemos usar `more /etc/passwd`, pero el ejemplo está basado a propósito en la manera Unix de hacer las cosas.

Los usuarios con experiencia en MS-DOS dirán que esto no es exclusivo de Unix, pero cabe aclarar en este punto que mientras que MS-DOS simula esta comunicación entre procesos utilizando archivos —ejecuta el primer programa guardando en un archivo la salida producida por éste y luego ejecuta el programa `more` sobre el archivo producido— en Unix esto ocurre en memoria y en tiempo real. Esto es que mientras que en MS-DOS tiene que terminar la ejecución del primer programa (dado que no es multi tareas) para poder ejecutar al siguiente programa con la salida del primero, en Unix, ambos están siendo ejecutados simultáneamente.

3.2 Redireccionamientos

Lo que acabamos de ver en la sección anterior se llama redireccionamiento de la salida. Esta es una base muy fuerte de la construcción de herramientas en Unix, el poder tomar la salida de un programa y dársela como entrada a otro. Veremos más adelante en la sección 5.1 la manera de aprovechar este método, de momento nos basta con conocer los principios bajo los que operan.

En lo que no aprendemos a usar un editor, tenemos una manera rápida de escribir a un archivo utilizando la redirección. Para escribir directamente a un archivo podemos hacer:

```
$ cat >  nombre_del_archivo
...
todo lo que queramos escribir sin posibilidad de regresarnos a la
l\'\i{}nea anterior
...
^D
```

El carácter `^D`[7] es el usado para significar fin de entrada.

La redirección '`>`' significa la salida mándala como entrada al archivo siguiente, no se utiliza cuando lo que le sigue es un comando o programa.

También existe la redirección '`<`' para tomar la entrada de un archivo. Por ejemplo, en lugar de `cat /etc/passwd` podríamos haber usado `cat < /etc/passwd`, pero esto es innecesario, ya que `cat` sabe como extraer el contenido de un archivo y no es necesario pedirle al sistema operativo que lo haga por él.

Recapitulando, sabemos que con `>` pasamos la salida de un comando a un archivo; con `<` tomamos el contenido de un archivo y se lo pasamos a un comando o programa y con `|` pasamos la salida de un programa a otro programa.

[7]En adelante para denotar los carácteres que se introducen a base de presionar al mismo tiempo la tecla control y otro carácter usaremos `^` y el carácter, así `^D` significa: presiónese simultáneamente control y D.

3.3 Navegando por el sistema de archivos

Hemos mencionado que el sistema de archivos es jerárquico y que podemos movernos dentro de él. Una de las herramientas indispensables para navegar en el sistema de archivos es `pwd` que nos dice en que lugar de la jerarquía nos encontramos. Por ejemplo, cuando el usuario Pablo entra en sesión, se encuentra inicialmente en su directorio "home", es decir en el directorio asignado a él dentro de la jerarquía de archivos. Si en ese momento ejecuta el comando `pwd` obtendrá:

```
$ pwd
/home/pablo
```

Digamos que ahora necesita escribir varias cartas y que éstas estén almacenadas de tal manera que pueda determinar directamente que son y a quién van dirigidas. De principio, bastaría con nombrarlas con el destinatario, pero pensemos que en un futuro la correspondencia aumentará y por cada persona tendrá varias cartas, y que además en su directorio almacenará otro tipo de información. Entonces, una buena solución es que tenga un directorio exclusivamente para almacenar las cartas, y que dentro de éste exista un directorio por cada destinatario. Así la estructura quedaría de ésta manera:

```
/home
        /pablo
                /cartas
                        /pedro
                        /gerencia
                        /asuntos_internos
                        /personal
                        /cobranzas
                        /comite_ejecutivo
```

Para crear un directorio se utiliza la instrucción `mkdir` de la siguiente manera, `mkdir {nombre del directorio}`[8]. Una vez creado, para posicionarnos dentro de él usamos `cd`. Así, la secuencia de instrucciones que Pablo necesita ejecutar para crear la estructura descrita es:

[8]En adelante usaremos {} para encerrar los argumentos a un programa o instrucción y [] cuando estos argumentos sean opcionales

```
$ mkdir cartas
$ cd cartas
$ mkdir pedro
$ mkdir gerencia
etc.
```

Para recorrer los directorios o cambiar el directorio actual se utiliza el comando `cd`. Se puede usar en forma relativa o absoluta. En el primer caso basta con dar el nombre de un subdirectorio a partir de donde estamos. Por ejemplo, si estamos en `cartas`, para cambiarnos al directorio `gerencia`, usaremos `cd gerencia`. Una vez allí, para irnos al directorio `cobranzas` podemos usar `cd ..` y `cd cobranzas`, con lo cual nos regresamos primero al directorio `cartas` y de ahí nos pasamos al directorio `cobranzas`. También es posible usar `cd ../cobranzas`. O incluso, para irnos de `cobranzas` a nuestro directorio de trabajo, podemos usar `cd ../..` ya que nos estamos moviendo al directorio padre del directorio padre del actual.

En el segundo caso, el de directorios asolutos, nos podemos mover dando la trayectoria completa. Por ejemplo, dentro del directorio `cobranzas` que se encuentra dentro de `cartas` podemos saber nuestra posición en el árbol con `pwd`: `/home/pablo/cartas/cobranzas` y movernos a nuestro directorio de trabajo con `cd /home/pablo`.

Existen tres maneras de regresarse al directorio original de trabajo. La primera es usando una variable de ambiente que tiene definido el valor de éste. Se usa de la siguiente manera: `cd $HOME`. La segunda es usando una pseudo variable que también tiene como definición a nuestro directorio de entrada: `cd ~`. La tercera es la más sencilla: simplemente usando `cd` sin ningún argumento, nos regresa a nuestro directorio "home" que es como se le conoce comunmente.

A la trayectoria que se ha de recorrer para llegar hasta un determinado directorio se le conoce como *path*. Así decimos que, en el caso del usuario Pablo, su *home* está en el *path* `/home/pablo`.

Ejercicio: ¿Qué pasa si damos: `cd .`; `cd ~/`; `cd ../.`**?**

Habrá ocasiones en que será necesario eliminar un directorio o un archivo. De hecho, para poder eliminar un directorio, debe de estar vacio o en otras palabras, no contener ningún archivo. Para borrar un archivo existe la instrucción `rm`, que entre sus opciones tiene dos que nos son de interés inmediato, `-i` y `-f`. Con `rm -i {nombre(s) de archivo(s)}` le indicamos que antes de borrar el archivo nos pida confirmación:

```
$ rm -i borrame
rm: remove ‘borrame’? y
```

Con `-f` forzamos el borrado de un archivo aún en situaciones especiales.

Para eliminar un directorio —que debemos de recordar que tiene que estar vacio— existe el comando `rmdir`. Veamos lo que pasa si tratamos de borrar un directorio que no está vacio. Primero examinamos el contenido de un supuesto directorio `lola` que digamos que existe en una máquina hipotética:

```
$ ls -al lola
total 29
-rw-rw-rw-   1   mancha  users   2243   Mar   16   02:39   carta.txt
-rw-rw-rw-   1   mancha  users   24987  Mar   16   02:39   intro.txt
```

Tiene dos archivos, tratemos de borrarlo:

```
$ rmdir lola/
rmdir: lola: Directory not empty
```

Ahora lo haremos borrando primero los archivos contenidos en él:

```
$ rm lola/*
```

Los nombres de los archivos en el directorio `lola` visto desde su padre, son: `lola/carta.txt` y `lola/intro.txt`. Al decirle que borre `lola/*` le decimos que borre todos los archivos contenidos dentro de `lola`. El carácter `*` es un comodín que significa cualquier carácter que se repite 0 o más veces. A diferencia de MS-DOS, el carácter “.” es uno más sin ningún significado particular.

Si ahora examinamos el contenido del directorio `lola`, vemos que sólo quedan dos directorios `.` y `..`, que siempre existen en todo directorio en Unix.

```
$ ls -al lola
total 2
drwxrwxrwx      2       mancha  users   1024    Mar     16      02:43   ./
drwxrwxrwx      4       mancha  users   1024    Mar     16      02:42   ../
```

Ahora podemos borrar este directorio sin ningún problema:

```
$ rmdir lola
```

Si trataramos de borrar el directorio `.` dentro de `lola`, obtendríamos lo siguiente:

```
$ rmdir lola/.
rmdir: lola/.: Operation not permitted
```

Ejercicio: ¿Se le ocurre la razón?

Instructor: Como puede verse, `lola/.` es una referencia cíclica a `lola`

Lo correcto es:

```
$ rmdir lola
```

De haber intentado borrar el directorio `lola/..` obtendríamos igualmente un error:

```
$ rmdir lola/..
rmdir: lola/..: Directory not empty
```

Ejercicio: ¿Porqué?

Instructor: Con ésta instrucción le pediríamos al sistema operativo que borre el directorio padre de `lola`, pero éste no está vacio, ya que al menos contiene a `lola`, así que no lo puede borrar

Habrá situaciones en que tenemos toda una parte de la estructura de archivos que necesitamos borrar y que sería toda una monserga ir borrando directorio a directorio por lo compleja que pueda ser dicha estructura. Digamos que a Pablo se le asigna una nueva función dentro de su organización y que ya no se va a dedicar a escribir cartas. En ese caso decide borrar el contenido del directorio `cartas` y todos sus subdirectorios con todos los archivos que estos contengan. Por lo que hemos visto hasta ahora, Pablo tendría que recorrer los subdirectorios hasta llegar a aquellos que están en la parte más baja de la estructura —es decir, que ya no contienen subdirectorios— y comenzar a borra desde allí hacia arriba. Afortunadamente existe una manera rápida de conseguir ésto, con la opción `-r` de `rm`.

¿Pero no era este un comando para borrar archivos y no directorios? Bueno, a final de cuentas un directorio es un archivo más dentro de la estructura de archivos de Unix, y por características propias (que están fuera de la intención de éste manual) es más conveniente hacerlo usando `rm` en lugar de `rmdir`.

Así la manera más rápida, y peligrosa, de borrar el directorio cartas y su contenido es:

```
$ rm -fr cartas
```

Nótese que no hubo necesidad de especificar las dos opciones por separado, es decir, `rm -f -r`; esto otra característica de los programas de Unix, cuando se utilizan juntas varias opciones, se pueden aglutinar todas con un sólo `-`.

Dijimos que era la manera más rápida y peligrosa. Lo primero es inmediato, `-f` asegura que en ningún momento `rm` se detendrá a preguntarnos si hace lo que le pedimos que hiciese. Lo segundo es un poco más difícil de explicar. Cuando se borra un archivo en Unix, se libera el espacio en disco que éste ocupaba y queda disponible para que el sistema operativo lo utilice en la creación de otro archivo. Como Unix es un sistema operativo multi tareas y multi usuarios, mientras nosotros borramos un archivo, puede ocurrir que el sistema operativo u otro usuario estén creando uno nuevo, de tal manera que de inmediato se ocupe el espacio donde estaba nuestro archivo. Esto quiere decir que es muy probable que en cuanto nosotros borremos el archivo, el área de disco que ocupaba sea utilizada para escribir otro. O sea, que una vez borrado quizá nunca más lo volveremos a ver. De hecho, Unix provee una utilería para tratar de recuperar información borrada de un disco, pero sin ninguna garantía de que se recupere la información borrada y además sólo puede ser ejecutada por el administrador del sistema. En la mayoría de los casos, es prácticamente imposible recuperar un archivo una vez que ha sido borrado.

3.3.1 Renombrado de archivos

Ahora bién, si lo que necesitamos es cambiar el nombre de un archivo, existe la instrucción `mv` cuya función real es *mover* un archivo de un lugar a otro. En particular, podemos mover un archivo de un nombre a otro. Esto es, si tenemos un archivo que se llama `clientes.morosos` y lo queremos renombrar como `clientes.hospitalizados`, basta con dar:

```
$ mv clientes.morosos clientes.hospitalizados
```

Ejercicio: ¿Se le ocurre una manera alterna de hacer esto?

Instructor: Una manera es `cp clientes.morosos clientes.hospitalizados` y después `rm clientes.morosos`

Si a `mv` se le da una lista de archivos y al final el nombre de un directorio, lo que hará es mover esos archivos al directorio especificado.

También se puede mover toda una estructura de archivos a otro directorio manteniendo dicha estructura.

3.3.2 Ligas a archivos

Ahora supongamos que dos usuarios, Alfonso Cruz y Elena Martínez, van a trabajar en el mismo proyecto y necesitan editar el mismo conjunto de archivos. Por conveniencia, ambos deberán pertenecer al mismo grupo. Supongamos que el directorio donde se encuentran los archivos con los que deben de trabajar se encuentra en el directorio de trabajo del jefe del área, llamado Pedro Enrique Armendárez. Sus cuentas se llaman `alfonso`, `elena` y `pedro` respectivamente y Alfonso y Elena pertenencen a los grupos: `usuarios, proyectos` y `inges`. El jefe a su vez, pertenece a los grupos: `jefes, inges, usuarios, proyectos` y `confidencial`. Estas tres personas estarán trabajando en un nuevo proyecto que se refiere a un puente. El administrador del sistema, una vez enterado, decide crear el grupo `puente` e incluye a `elena, pedro` y `alfonso` en este grupo, de tal manera que ahora Pedro Enrique en su directorio de trabajo crea el directorio `proyecto.puente`, le cambia el grupo con `chgrp` a puente:

```
cd
cd trabajo
mkdir proyecto.puente
chgrp puente proyecto.puente
chmod u=rwx proyecto.puente
chmod g=rwx proyecto.puente
chmod o-rwx proyecto.puente
```

de tal manera que sólo él y los pertenecientes al grupo `puente` pueden ver y modificar el contenido de dicho directorio. Ahora, cada vez que alguno de ellos cree un nuevo archivo (o directorio) dentro del directorio, deberá cambiarle el grupo a `puente` y tener cuidado que los permisos sean los pertinentes.

Pero, ¿no será demasiada lata que Elena y Alfonso tengan que recorrer toda la estructura cada vez que necesitan trabajar en éste directorio? La primera respuesta que nos viene a la mente es que no hay problema, sabemos perfectamente cómo hacerlo. Pero también hay una manera más fácil: las ligas de archivos.

Todo será más fácil si Elena y Alfonso dentro de su directorio `trabajo` tienen un subdirectorio llamado `proyecto.puente` que es idéntico permanentemente al que está en el directorio de Pedro Enrique. Para esto existe la instrucción `ln` que construye ligas de un archivo o directorio a otro.

Como Elena y Alfonso tienen acceso al directorio absoluto `/home/pedro/trabajo/proyecto.puente`, basta con que en su respectivo directorio `trabajo` cada uno haga lo siguiente:

```
cd
cd trabajo
ln /home/pedro/trabajo/proyecto.puente proyecto.puente
chgrp puente proyecto.puente
```

y como éste directorio —en el directorio original, `/home/pedro/trabajo/proyecto.puente`— ya tiene los permisos adecuados, no necesitarán hacer nada mas.

3.3.3 Cómo revisar archivos

Así como tenemos la instrucción `more` para revisar el contenido de un archivo en pantalla de una manera pausada, tenemos las instrucciones `head` y `tail` que permiten examinar nada más el principio y el final de un archivo respectivamente. Ambos pueden llevar como argumento el número de líneas que se desean examinar. Por ejemplo, para examinar las primeras cinco líneas y las últimas tres líneas de un archivo de datos, utilizamos:

```
$ head -5 datos
090145241|121 22:58401|105106122113109109093101001001001002072071
090145242|121 22:46400|109107107105130126124126000000001000050051
090145282|121 22:46480|142143162144168158127148002003003002039045
090145291|121 22:26388|094093098101125122108108002002001001065062
090145292|121 22:24387|099123114118117092086092001003004002070070
```

```
$ tail -3 datos
201022211|122 00:11785|05504304500010411312000000400600400000950 91
201022231|222 00:03368|00300300200007107107100000100200200002902 5
201022233|122 00:24  0|00500500500014513914700000000000000002802 5
```

Ahora supongamos que queremos ver de la línea 496 a la línea 500 del mismo archivo:

```
$ head -500 datos | tail -5
091111024|121 20:40  0|097028000000120041000000000100000000000510 14
091111032|121 23:37429|115109115115175181173167000002002000049044
091111041|121 23:25680|202197192191236236228221003002006004088085
091111052|121 23:51446|116110102112174178183170001002001001037039
091111062|121 23:34504|128114119134178186178171001001002001064001
```

¿Y cómo sabemos cuántas líneas tiene el archivo? Con la instrucción `wc`, abreviatura de *word counter*. De hecho, `wc` nos dá mas información que el número de líneas del archivo:

```
$ wc datos
  27588   55941 5231568
```

donde cada columna nos dice el número de líneas, palabras[9] y carácteres. En éste caso sólo nos interesa saber el número de líneas, así que repetimos el ejemplo con el modificador `-l`:

```
$ wc -l datos
  27588
```

De igual manera, si sólo queremos saber el número de palabras o de carácteres utilizaríamos los modificadores `-w` y `-c` respectivamente.

[9]En éste caso el número de palabras se refiere a cualquier carácteres alfanuméricos delimitados por signos de puntuación, incluyendo al espacio, tabulador y cambio de línea.

3.3.4 Impresión de archivos

Para imprimir un archivo existe la instrucción `lpr` que además de enviarlo a la impresora, lo *pagina* o separa por páginas y le añade a cada una una *cornisa* o encabezado con el nombre del archivo y número de página. Tiene varias opciones para indicarle a que impresora se envía, cuantas líneas por página permite la impresora o se desean, que debe de ir en la cornisa y otras opciones más que deberán ser examinadas en la página de manual, ya que de de una implementación a otra cambian algunos parámetros. En éste caso, lo más seguro es preguntar al administrador del sistema los nombres de las impresoras y las opciones que permite `lpr` de acuerdo a éstas.

3.4 Espacio en disco

En ocasiones es necesario saber cuanto espacio ocupa todo un directorio y cuanto espacio queda libre en un disco. Existen dos instrucciones para esto, `du` y `df`. `du` nos dice el espacio ocupado en kilobytes dentro de un directorio si se ejecuta sin ningún argumento:

```
$ du
15      ./user-alpha-4/emacs-chapter-stuff
157     ./user-alpha-4/ps-files
2600    ./user-alpha-4
2634    .
```

En este caso, el directorio `./user-alpha-4/emacs-chapter-stuff` ocupa 15 kilobytes, `./user-alpha-4/ps-files` 157 Kb, `./user-alpha-4` 2600 Kb y todo el directorio actual junto con sus subdirectorios ocupa 2634. La salida de du muestra los nombres de los subdirectorios con el prefijo `./` para indicar que es a partir del directorio actual.

Podemos pedir también el espacio ocupado por un directorio por su nombre absoluto:

```
$ du /etc
349     /etc
```

La salida de `df` es un poco más críptica:

```
$ df
Filesystem      1024-blocks    Used    Available   Capacity   Mounted on
/dev/hda1       237133         208827  16450       93%        /
/dev/hdb        208879         194597  3839        98%        /home
```

La primera columna nos indica el *filesystem* o sistema de archivos, o dispositivo físico, o en términos más comprensibles, el disco. La segunda nos dice el espacio total en el disco en unidades de 1024 bytes, o 1 kilobyte, aunque las unidades varían de una implementación de Unix a otra. La tercera y cuarta columnas indican el espacio utilizado y el espacio disponible en las mismas unidades que la segunda. La quinta columna indica el porcentaje ocupado del disco. Por último, la sexta columna indica en que parte del sistema de archivos está *montado* el disco, así que si deseamos saber cuanto espacio queda disponible para el usuario Pablo, debemos considerar en que lugar del sistema de archivos está el directorio donde trabaja. Como está en `/home/pablo` en éste caso, sabremos que el espacio que le queda disponible, son 3 839 Kb o 3.7490 Mb, dado que `pablo` es un subdirectorio dentro de `home`.

Cabe hacer la aclaración que Unix provee de mecanismos para acotar el espacio en disco disponible para cada usuario, pero la mayoría de las veces esta restricción no se aplica, con lo cual si hay cinco usuarios en el sistema, estos cinco usuarios compiten por el espacio disponible en el disco donde estan montados sus directorios.

La elección de restringir el espacio en disco a cada usuario está determinada por las políticas de uso que emplee el administrador del sistema.

Capítulo 4

Editores

4.1 Editor de pantalla completa `vi`

El editor más frecuente en Unix es `vi`. Es un editor que trabaja línea a línea y que muestra una pantalla de texto a la vez.

Para inciar una sesión de edición, se ejecuta el programa `vi` seguido del nombre del archivo a editar, y dado el caso también la trayectoria. Por ejemplo, para editar el arvhivo `/tmp/borrame`, basta con dar `vi /tmp/borrame`, o para editar un archivo en el directorio actual: `vi borrame`.

Si el archivo no existe `vi` lo crea. De igual manera, podemos simplemente invocar a `vi`, comenzar a escribir y despúes nombrar el archivo al momento de salvarlo.

Al ser ejecutado `vi`, presenta una pantalla con el texto del archivo, y las líneas después del final del archivo aparecen con el carácter `~` para indicar que a partir de ahí el archivo está vacio. Obviamente, si comenzamos a editar un archivo nuevo, todas las líneas aparecerán con éste carácter.

Tiene tres modos de trabajo. El modo de *inserción*, el de *edición* y el de *comandos*.

En el modo de inserción, toda la entrada que demos en el teclado se inserta en el archivo en el punto donde se encuentre el cursor. En el modo de edición daremos instrucciones que alteran el contenido, como por ejemplo para posicionarse en determinado punto, hacer reemplazos de texto, copiar o mover bloques de texto, etc. En el modo comandos se dan instrucciones para salvar el archivo, traer a edición otro, insertar otro archivo en el punto donde se está, terminar la edición, etc.

En el modo edición, las instrucciones para mover el cursor en del texto son:

comando	se desplaza:
`l`	un espacio a la derecha
`h`	un espacio a la izquierda
`j`	una línea hacia abajo
`k`	una línea hacia arriba
`$`	al final de la línea
`^`	al principio de la línea
`w`	a la siguiente palabra
`e`	al final de la palabra
`b`	al principio de la palabra
`)`	al final de la frase
`(`	al inicio de la frase
`{`	al inicio del párrafo
`}`	al final del párrafo
`n|`	a la columna n-ésima
`H`	a la primera columna de la primera línea de la ventana
`L`	a la primera columna de la última línea de la ventana
`nG`	a la primera columna de la n-ésima línea del archivo

Para el control de la parte del texto que se despliega en la pantalla:

instrucción	acción:
`^d`	desliza el texto hacia arriba
`^u`	desliza el texto hacia abajo
`^f`	despliega la ventana de texto siguiente
`^b`	despliega la ventana de texto anterior
`^l`	redespliega el texto en la ventana actual

Las instrucciones para borrar texto:

`dw`	suprime la palabra donde está el cursor
`dd`	suprimir la línea donde está el cursor
`D`	borra el texto entre el cursor y el fin de la línea
`x`	borra el carácter sobre el que esta el cursor

Para hacer reemplazos de texto:

`cw`	cambiar la palabra actual
`cc`	cambiar la línea actual
`C`	cambiar desde el cursor hasta el final de la línea
`r`	cambiar el carácter sobre el que está el cursor

Algunas instrucciones suplementarias:

`u`	anular la última instrucción dada
`/`	realiza una búsqueda hacia adelante
`?`	realiza una búsqueda hacia atrás
`n`	busca la siguiente ocurrencia de la última búsqueda
`.`	repite la última instrucción
`Y`	extrae la línea
`p`	se coloca en la línea de abajo
`P`	se coloca en la línea de arriba
`ZZ`	salva el archivo y termina la edición
`ESC`	cancela una orden
`:`	se cambia a modo comandos

Cuando `vi` esta en modo de edición, para cambiarse a modo inserción se hace con el carácter `i` y se posiciona antes del cursor y con `a` después de éste. Con `Esc` se cambia a modo edición de nuevo.

En modo de comandos, se tienen las siguientes funciones:

`:w`	salva el archivo en el disco
`:q`	abandona la edición sin guardar los cambios
`:wq`	escribe y termina
`:q!`	abandona sin escribir cuando se realizó algún cambio.
`:r`	carga otro archivo
`:e`	edita el archivo
`:f`	cambia o dá nombre al archivo actual
`:n`	se posiciona en la n-ésima línea
`:Esc`	se pasa a modo de edición

`vi` tiene muchas más instrucciones y es capáz de realizar tareas muy complejas. Es importante conocer la mayoría de ellas para hacer más eficiente nuestro trabajo, pero son demasiado extensas para incluirlas todas en este curso. Existen libros especializados donde se describe con toda sobriedad esta poderosa herramienta.

Existen además de `vi` otros editores para Unix, incluso más poderosos que él, pero esto depende de cada implementación de Unix. `vi` y `ex` (que por ser algo más limitado, no trataremos aquí) son los únicos que se garantiza que se pueden encontrar en cualquier instalación de Unix. Entre los editores más populares se encuentra Emacs, pero cuenta con un conjunto de instrucciones realmente complejo, lo cual forma parte de su tradición de ser para usuarios avanzados.

Haremos ahora un ejemplo para familiarizarnos con `vi`. Comenzaremos editando un archivo nuevo que se llamará `cordero.txt`. Invocamos a `vi` con este argumento,

```
$vi cordero.txt
```

ahora la pantalla se limpia y aparecen varios renglones con tildes en el márgen izquierdo y en la parte inferior izquierda aparece el mensaje `¨cordero.txt¨ [New file]`, que indica que se esta creando un archivo nuevo.

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

```
~
~
~
~
~
~
 "cordero.txt" [NEW FILE] 1 line, 1 char
```

En este momento nos encontramos en el modo de edición con el cual podemos iniciar la captura del texto. Para esto damos la instrucción `i` y nos pasamos a modo inserción. Ahora podemos escribir cualquier texto que necesitemos.

Cuando se tengan un par o más párrafos podemos ejercitar algunas de las instrucciones para movernos por el texto y para realizar búsquedas e incluso reemplazos.

Nuevamente, para tener un conocimiento más profundo de `vi` es recomendable ver la *página* de `vi` con `man vi`.

4.2 Editor de línea `ed`

El editor `ed` fue hecho con la idea de tener un editor rápido y pequeño con lo mínimo indispensable. Es, además, un editor confiable y que puede ser usado en las peores condiciones: con terminales lentas, en conexiones por modem y, quizá la más interesante, desde archivos por bloques. La mayoría de los editores asumen que toman la entrada directa desde el teclado y que controlan una terminal, por esto no pueden ser empleados como `ed`.

En su modo de operación normal, trabajan sobre una copia del archivo, para sobreescribirlo, hay que dar una instrucción específica. Trabaja ssobre una línea o un grupo de líneas que cumplan con un patrón. Cada comando es un solo carácter, típicamente una letra. Cada comando puede ser precedido por uno o dos números de línea, que indican la línea o el rango de líneas al que serán aplicados. De no ser dado el número de línea, actúa sobre la actual. Veamos un ejemplo:

```
$ ed
a
Cuando abandonaste el cielo
```

```
?Donde dejaste tus alas?
Cuando abandonaste el cielo
Quisiera saberlo, ?perdiste el halo?

?Como encontraste el camino?
?Aun te escondes cuando sale el sol?
Cuando abandonaste el cielo
?Perdiste las alas?
.
w angel
233
q
$
```

Después de invocar a `ed` inmediatamente le pedimos que añada el texto con el comando `a` y comenzamos a guardar el texto de un pequeño poema. Terminamos poniendo un . como único carácter en la línea y después le pedimos que salve el archivo con el nombre de `angel`. `ed` nos responde con el número de carácteres que contiene, y terminamos la sesión de edición con una `q`.

Si quisieramos añadir unas cuantas líneas más, podríamos hacerlo con:

```
$ ed angel
233
a

Ahora veo a un largo tren venir despacio
y una muerte agradecida que me reconforta
dime Reina Ines ?vendras a verme?
El amigo del diablo es amigo mio
.
q
?
w
384
q
$
```

En esta ocasión, invocamos a `ed` directamente con el nombre de un archivo que ya existe, así que nos responde con el número de carácteres que contiene el archivo. Comenzamos a añadir unas líneas, nos detenemos y tratamos de salirnos. Esta vez `ed` nos avisa con `?` que el archivo no ha sido salvado. No debemos esperar una comunicación más comprensible de `ed`. Su manera de avisarnos que algo no le gusta o que las cosas no van bien es con un críptico `?`. En este caso una segunda `q` le indicaría que realmente nos queremos salir sin salvar los cambios que hicimos. En todo momento una `Q` le indica a `ed` que deseamos terminar la sesión sin guardar los cambios.

Cuando queremos revisar el texto ya escrito, le podemos pedir que imprima las líneas en un cierto rango con la instrucción `a,bp` donde `a` y `b` son números de línea. Por ejemplo, para ver la segunda estrofa:

```
$ ed angel
384
6,9p
?Como encontraste el camino?
?Aun te escondes cuando sale el sol?
Cuando abandonaste el cielo
?Perdiste las alas?
.,$p
?Perdiste las alas?

Ahora veo a un largo tren venir despacio
y una muerte agradecida que me reconforta
dime Reina Ines ?vendras a verme?
El amigo del diablo es amigo mio
q
$
```

Los carácteres `.` y `$` tienen el significado especial de ser la línea actual y la última línea respectivamente como podemos ver en la segunda parte del ejemplo.

Si damos un enter por si sólo o un `-` seguido de un enter, nos lista la siguiente línea y la anterior respectivamente, moviéndose hacia adelante y hacia atrás en el archivo. No permite moverse más allá de la primera y de la última línea, ni imprimirlas en orden inverso.

Podemos hacer búsquedas utilizando los operadores `/patron/` y `?patron?` hacia adelante y hacia atrás respectivamente. Una vez que encontramos el primero, podemos traer el siguiente con las formas `//` y `??`.

```
$ ed angel
384
/con/
?Como encontraste el camino?
//
?Aun te escondes cuando sale el sol?
//
y una muerte agradecida que me reconforta
?alas?
?Donde dejaste tus alas?
??
?Perdiste las alas?
q
$
```

Y podemos usar una búsqueda como parte de un rango:

```
1,/saber/p
Cuando abandonaste el cielo
?Donde dejaste tus alas?
Cuando abandonaste el cielo
Quisiera saberlo, ?perdiste el halo?
.-1,.+3p
Cuando abandonaste el cielo
Quisiera saberlo, ?perdiste el halo?

?Como encontraste el camino?
?Aun te escondes cuando sale el sol?
```

Y como vemos, también podemos usar el `.` como ancla e imprimir de la línea anterior a tres más adelante de la actual.

La forma genérica de los comandos es número de línea o rango seguido del comando. Para añadir a partir de una línea a partir de línea se usa

#a, para insertar antes de la línea #i. Para borrar el rango de líneas de la i-ésima a la j-ésima `i,jd` y `i,jc` para reemplazar el rango de líneas por las que se dan a continuación.

Para hacer reemplazos en base a patrones se utiliza `s/viejo/nuevo/`:

```
2p
?Donde dejaste tus alas?
2s/alas/celtas/
2p
?Donde dejaste tus celtas?
```

Después del cambio se puede usar el modificador `g` para que haga el reemplazo en todas las ocurrencias de la línea o en todas las ocurrencias en el rango de las líneas: `s/viejo/nuevo/g`. Por supuesto que la sintáxis es extensible a rangos delimitados por patrones de búsqueda: `4,9s/viejo/nuevo/`, `1,$s/viejo/nuevo/`, `1,/viejo/s/viejo/nuevo/`, etc.

Cuando necesitamos incluir el patrón viejo dentro del nuevo, no necesitamos reteclearlo, el carácter `&` toma el valor:

```
2p
?Donde dejaste tus alas?
2s/alas/& marchitas/p
?Donde dejaste tus alas marchitas?
```

Y como vemos, la `p` después del comando de reemplazo imprime la línea.

Para copiar y mover bloques, tenemos las instrucciones `i,jtk` `i,jmk` donde el primero copia las líneas en el rango de la i-ésima a la j-ésima después de la línea k-ésima. Igual en el segundo caso, pero borrandólas de su posición original.

También podemos incluir un archivo en el actual con la instrucción `r` en la forma `nr archivo` con la cual lo insertamos en el actual a partir de la línea n-ésima.

Con `i,jw archivo` copiamos de la línea i-ésima a la j-ésima en el archivo denominado. Con `i,jW archivo` copiamos de la línea i-ésima a la j-ésima *al final* del archivo denominado.

Capítulo 5

Interface con el usuario

En la interacción con Unix se puede, aparte de hacer las mismas tareas comúnes que en cualquier otro sistema operativo, tener la posibilidad de *programar* o automatizar tareas de una manera realmente eficiente.

En la sección 3.1 anterior dejamos pendiente la explicación del programa `grep`. Vagamente recordamos que se trata de un programa que busca cadenas de texto dentro de un archivo o de varios. Bien, con frecuencia nos enfrentamos a éste problema: necesitamos imprimir o alterar un archivo en particular, del cual no recordamos el nombre, pero si que contenía una palabra o una frase en particular. Digamos que recordamos que se trata de una carta que enviamos a un cliente importante donde usamos la frase: "es usted un cretino insolente". La manera pedestre de encontrar el archivo sería revisar todos y cada uno de los archivos contentidos en el subdirectorio `cartas` del directorio `clientes`. Pero, ¿y si son quinientas cartas? Bueno, en este caso comprenderemos porque `grep` es valioso. Basta con dar:

```
pablo@diva~/cartas/clientes$ grep "es usted un cretino insolente" *
director.Banco.Nacional.25.de.junio.de.1994.txt:si cree usted que
podr\'a salirse con la suya y no pagarnos el adeudo, me tomo la libertad de
informarle que es usted un cretino insolente,
director.regional.Nuevo.Leon.27.junio.1994.txt:y sin
ningun motivo aparente, me espet\'o: el cretino insolente lo ser\'a
usted. Obra en mi poder la prueba
```

Y vemos como `grep`, rápidamente, encontró dos documentos con la frase que entre comillamos. La razón del entre comillado, es que `grep` espera que el primer argumento sea el elemento a buscar y los demás sean los archivos donde buscará, así que con las doble comillas le avisamos al sistema operativo que la frase forma un grupo y que éste deberá ser pasado al programa `grep` como un sólo argumento.

El segundo argumento, el asterisco, es en realidad un *wildcard* que le avisa al sistema operativo: toma todos los archivos de éste directorio y pásaselos como argumentos al programa. Así que, en realidad, `grep` no recibe `*` como segundo argumento, sino toda la lista de archivos que contiene el directorio. Esta es otra diferencia con respecto a MS-DOS en que, como todo programador sabrá, el programa recibiría en efecto el asterisco y el programa debe saber que hacer con él.

¿Qué pasaría si ni siquiera sabemos en que subdirectorio se encuentra la carta? Bueno, pues desde el subdirectorio `cartas` podríamos pedir:

```
pablo@diva~/cartas$ grep "es usted un cretino insolente" */*
./clientes/director.Banco.Nacional.25.de.junio.de.1994.txt:si
cree usted que podr\'a salirse con la suya y no pagarnos el adeudo, me
tomo la libertad de informarle que es usted un cretino insolente,
./clientes/director.regional.Nuevo.Leon.27.junio.1994.txt:y
sin ningun motivo aparente, me espet\'o: el cretino insolente lo ser\'a
usted. Obra en mi poder la prueba
```

¿Puede explicar que ocurre al utilizar `"*/*"` como último argumento?

Instructor: En este caso, le pedimos al sistema operativo que le pase como argumentos a `grep` todos los archivos de todos los subdirectorios que se encuentran a partir del actual.

Ahora bien, supongamos que conocemos el nombre del archivo, pero no recordamos en que directorio está. Digamos que el nombre del archivo es `breve_carta_donde_le_recuerdo_a_mi_tia_Genoveva_que_soy_su_sobrino_favorito.txt`. De nuevo sabemos que está en algún lugar del árbol a partir del subdirectorio `cartas`. De nuevo, `grep` nos puede ayudar.

```
pablo@diva~/cartas$ ls -R | grep breve_carta_a_mi_tia
./herencias/breve_carta_donde_le_recuerdo_a_mi_tia_Genoveva_que_soy_su_sobrino_favorito.txt
```

La opción `-R` de `ls` lista recursivamente el contenido de archivos y subdirectorios.

O podríamos utilizar la opción `-a` de `du`, que nos lista el espacio ocupado en bloques de todos los archivos:

```
pablo@diva~/cartas$ du -a | grep breve_carta_a_mi_tia
423     ./herencias/breve_carta_donde_le_recuerdo_a_mi_tia_Genoveva_que_soy_su_sobrino_favorito.txt
```

Y que además nos informa que el archivo contiene 423 kilobytes.

En realidad no necesitamos recurrir a un *pipe* para hacer esto. Existe el programa `find` que se utiliza para encontrar archivos y que además permite ejecutar acciones sobre de ellos. En este caso lo podríamos haber utilizado de la siguiente manera:

```
pablo@diva~/cartas$ find . -name '*Genoveva*favorito*' -print
./herencias/breve_carta_donde_le_recuerdo_a_mi_tia_Genoveva_que_soy_su_sobrino_favorito.txt
```

Con el primer argumento '.' le informamos a `find` que la búsqueda se realizará a partir del directorio actual, con `-name` le pedimos que haga la búsqueda por nombre, con el tercero le informamos que como parte del nombre aparecen `Genoveva` y `favorito`, en ése orden, y con el cuarto le pedimos que cuando lo encuentre, imprima el *path* donde lo halló. Utilizamos la construcción `'*Genoveva*favorito*'` por la siguiente razón, con los asteriscos le informamos que antes de `Genoveva` ocurre cualquier combinación de carácteres, al igual que entre `Genoveva` y `favorito`, así como después de éste último. Utilizamos el apóstrofe para ocultar los astericos al *shell*, que es el programa con el que hemos estado interaccionando todo este tiempo.

Ejercicio: De no ocultar los asteriscos al *shell* ¿qué ocurriría?

Instructor: El *shell* trataría de resolver el nombre del archivo como si se encontrase en el directorio actual. Al no ser así, `find` no encontraría jamás el archivo aún cuando sabemos que éste existe.

Como hemos visto hasta ahora, Unix nos proveé de varias maneras de hacer las cosas y las construcciones que emplean los *pipes* nos permiten llevar a cabo tareas tan complejas como se nos ocurra.

Haremos un ejemplo un poco mas complicado que nos revela cúan poderoso puede llegar a ser Unix. En éste caso sabemos que en algún lugar de nuestro directorio existen varios archivos que como parte del nombre tienen la palabra `pericles` y queremos encontrar en particular aquél que habla de `parias nucleares`. Como ya sabemos usar `find` para encontrar archivos que cumplen con un patrón determinado y además sabemos utilizar `grep` para encontrar frases dentro de archivos, la solución es sencilla.

```
pablo@diva~$ grep 'parias nucleares' `find . -name '*pericles*' -print`
./textos/poesia/oda_postmoderna_a_pericles.txt:las mujeres
sucias, parias nucleares de \'esta tragedia
```

Sabemos bien de que se trata lo de `grep 'parias nucleares'` y también comprendemos que queremos al utilizar `find . -name '*pericles*'`

`-print`, lo único nuevo en este caso es la construcción tan extraña y la comilla simple que encierra a `find` y sus argumentos. Sabíamos que el primer argumento a `grep` es la palabra o frase a buscar dentro del archivo y que los demás argumentos son archivos. También sabíamos que `find` encuentra archivos que cumplen con una condición determinada. Entonces, ¿por qué no aprovechar ambos para realizar la tarea? Para conseguirlo, le indicamos al *shell* que el segundo argumento a `grep` va a ser el resultado del `find`. La comilla antes y después son precisamente para esto. El *shell* al encontrar una instrucción encerrada por comilla sencilla, sabe que primero debe de ejecutar esa instrucción y luego el resto del comando. En este caso ejecutará primero el `find` y el resultado lo utilizará como argumento al `grep`.

Ahora vamos a hacer un corrector ortográfico para pobres. Lo primero que necesitamos es un diccionario. Para esto, tomamos algunos archivos que sepamos con seguridad que no contienen muchas faltas de ortografía. Los pegamos todos a un archivo temporal con el programa `cat`:

```
pablo@diva~$ cat carta.txt entrevistas.virtuales.txt ipt.txt \
pgp.txt wired.txt > /tmp/borrame
```

Construimos el diccionario:

```
pablo@diva~$ cat /tmp/borrame | tr -c 'a-zA-Z\341\351\355\363\372\361' ' '| \
tr ' \t' '\n'| sort | uniq > diccionario
```

Y con esto ya tenemos una lista de palabras, una por renglón, que podemos utilizar para el corrector. Veamos por partes como se construye. Hacemos un `cat` y un `pipe` para pasarle el archivo con que vamos a construir el diccionario a un programa llamado `tr`. Para comprender mejor que es lo que hace `tr`, veamos primero lo que occurre con la segunda aparición de éste programa en la línea de comandos. La tarea de `tr` es la de traducir conjuntos de carácteres en conjuntos de carácteres en los ordenes respectivos. Si queremos traducir todo un archivo de minúsculas a mayúsculas, bastaría con usar `tr 'a-z' 'A-Z'`. Por ejemplo, si tenemos un archivo llamado `minusculas` y queremos pasar todo su contenido a mayúsculas en el archivo `salida.mayusculas`, usaríamos:

```
pablo@diva~$ cat minusculas | tr 'a-z' 'A-Z' > salida.mayusculas
```

Regresando al ejemplo del corrector ortográfico, para construir la lista de palabras necesitamos que estas ocurran una por línea y además que no aparezcan signos de puntuación, números ni símbolos. Lo que hacemos en el ejemplo con la primera ocurrencia de `tr` es, precisamente, eliminar todos los carácteres que no sean alfabéticos aprovechando la opción `-c` que indica que se tome el complemento del primer conjunto, es decir que se traducirán todos los carácteres que no aparezcan en el primer conjunto, que son todos los carácteres distintos de la `a` a la `z` minúsculas y mayúsculas y seis números que representan a las vocales acentuadas y a la `ñ` en el estándar ISO-latin1.

Una vez hecho lo anterior, en la *tubería* en lugar del archivo original, viaja un archivo donde no hay signos de puntuación, dígitos, símbolos ni fines de línea. Sólo queda una lista de palabras separadas por espacios, y nosotros la necesitamos separadas una por línea, así que recurrimos nuevamente a `tr` con los conjuntos `'␣'` y `'\n'`. En este caso el primer conjunto es `'␣'`, que consiste en un espacio y el segundo es un identificador especial que representa al cambio de línea. Así que ahora en la tubería viaja un archivo que tiene una palabra por línea.

Por último, necesitamos que las palabras estén en orden y que no aparezcan más de una vez. Para esto usamos `sort`, que se encarga de ordenarlas, y `uniq`, que deja pasar sólo una ocurrencia de cada palabra.

Estas conversiones las hicimos porque vamos a utilizar una utilería de Unix llamada `comm`, que compara línea por línea dos archivos ordenados y en la salida escribe en la primera columna lo que aparece exclusivamente en el primer archivo, en la segunda lo que ocurre exclusivamente en el segundo y en la tercera lo que aparece en ambos. Digamos que tenemos dos archivos llamados `Juan` y `Pedro`, que representan a dos matemáticos amigos nuestros, y que cada archivo contiene la lista de coches que posee cada uno. Así, si Juan tiene: un Bentley, un Ferrari, un Jaguar, un Masserati, un Peugeut y un Roll Royce, y Pedro tiene: un Barmann Costa, un Jaguar, un Lamborghinni, un Mercedes Benz, un Peugeut y un Porsche; entonces la salida será:

```
pablo@diva:~$ comm Juan Pedro
                Barmann Costa
Bentley
Ferrari
                                Jaguar
```

```
                Lamborghinni
Masserati
                Mercedes Benz
                                        Peugeut
                Porsche
Roll Royce
```

Donde vemos que en efecto, ambos coinciden en tener un Peugeut y un Jaguar, aunque naturalmente, en la vida real, tratándose de personas de gran éxito económico como los matemáticos, ésta lista sería demasiado grande para hacer la comparación a mano, lo cual justificaría aún mas el utilizar programas como `comm`.

Un último detalle, es que como salida del corrector ortográfico queremos que se produzca la lista de palabras que tenemos en el archivo a revisar y que no están en el diccionario, así que sólo necesitamos como salida la tercera columna producida por `comm`. Esto lo podríamos hacer utilizando otro programa, pero afortunadamente `comm` considera este caso y como opción se le puede decir que columnas se omiten. Esto se hace sencillamente indicándoselas con el número de columna, en nuestro ejemplo omitiremos las columnas segunda y tercera, es decir, las columnas que listan las palabras en nuestro documento y las del diccionario. Así que la opción que le daremos a `comm` es `-23`. Otra aclaración es que `comm` espera como entrada dos archivos y los que vamos a utilizar son el diccionario que creamos, del mismo nombre, y el archivo que está viajando por la *tubería*, el cúal no existe como archivo con un nombre asignado, pero ésto tampoco es problema, porque en Unix los *archivos* creados en base a *pipes* tienen como nombre alternativo simplemente `-`. Así que finalmente estamos listos para ejecutar el corrector ortográfico de los pobres:

```
pablo@diva~$ cat unix.txt | tr -c 'a-zA-Z\341\351\355\363\372\361' ' '| \
tr '\t' '\n'| sort | uniq | comm -23 -diccionario | more
```

De haber seguido con atención nos daremos cuenta de lo limitado que resulta este corrector por varias razones. La primera es que no nos indica en que lugar del archivo ocurre la palabra, otra es que no hay ninguna garantía de que esa palabra esté mal, simplemente no está entre las que consideramos correctas y la tercera es que no hace el reemplazo adecuado, pero precisamente por eso lo llamamos el corrector ortográfico de los pobres.

5.1 Archivos de comandos

Ahora veamos la manera de no tener que escribir cada uno de los comandos cada vez que queramos hacer algo similar. La mejor manera es escribir en un archivo estos comandos y hacer a éste archivo ejecutable de tal manera que baste con invocarlo con los parámetros adecuados para que todo funcione como deseamos.

Bueno, el caso es que siguiendo la costumbre Unix de asignar nombres crípticos a los programas y comandos, llamaremos `cdc` al archivo que crea diccionarios y `co` al corrector ortográfico. El contenido del archivo `cdc` es el siguiente:

```
cat $* > /tmp/borrame
cat /tmp/borrame | tr -c 'a-zA-Z\341\351\355\363\372\361' ' ' | \
tr ' \t' '\n' | sort | uniq > diccionario
rm -f /tmp/borrame
```

donde `$*` significa, como vimos en la secciónr̃efsec:variables, todos los parámetros que se le den a un archivo ejecutable en la línea de comandos.

Y el contenido de `co`:

```
cat $1 | tr -c 'a-zA-Z\341\351\355\363\372\361' ' ' | tr ' \t' '\n' | \
sort | uniq | comm -23 - diccionario | more
```

donde `$1` significa el primer parámetro que recibe en la línea de comandos.

5.2 Variables

Habrá ocasiones en que necesitaremos almacenar un valor o una cadena de carácteres al realizar alguna operación con el *shell*, para después pasarle éste valor a otro programa o, simplemente, para después examinarlo. El *shell* proveé de una manera de realizar esto por medio de las *variables de entorno*, también llamadas *variables de ambiente*.

Estas variables existen únicamente durante la ejecución del *shell*. Así, si en una sesión creamos una variable o modificamos su valor, al terminar la sesión éste se pierde.

Para asignar un valor a una variable, basta con poner el nombre de la variable seguido de un signo de igual y el valor:

```
$ variable=valor
```

El nombre de la variable precedido por el carácter `$` es el valor de la variable. La mejor manera de examinar el valor es utilizando el comando `echo`:

```
$ color=rojo
$ echo $color
rojo
$ telefono=5552314
$ echo $telefono
5552314
$
```

Cuando se necesita asignar a una variable una cadena que incluye espacios o carácteres que normalmente el
em shell interpretaría, es necesario encerrar el valor entre comillas sencillas:

```
$ hacer='tengo que pasar al super; ir a la tintorer\'\i{}a; hablarle a mi tia'
$ echo $hacer
tengo que pasar al super; ir a la tintorer\'\i{}a; hablarle a mi tia
$
```

El *shell* tiene interconstruidas ciertas variables que son importantes para su funcionamiento. Por ejemplo la variable `PATH` contiene la lista de directorios donde serán buscados los programas a ejecutar. Esta variable se define en el archivo de configuración del *shell* actual[1]. Por ejemplo, lo más usual es que esta variable esté prefijada con los siguientes valores:

```
$ echo $PATH
:/bin:/usr/bin
$
```

pero sabemos que existe el directorio `/usr/games` donde se encuentran algunos juegos. En particular nos interesa ejecutar el programa llamado `fortune` que, `man` lo sabe, produce como salida una frase simpática. Pero acabamos de examinar el contenido de la variable `PATH` y claramente podemos observar que el directorio `/usr/games` no está incluido. Bueno ya sabemos como incluirlo:

[1]Ver sección 5.3.

```
$ echo $PATH
:/bin:/usr/bin
$ PATH=$PATH:/usr/games
$ echo $PATH
:/bin:/usr/bin:/usr/games
$
```

y como podemos observar los directorios van separados por dos puntos, :, esta es la manera tradicional para las variables que contienen varios directorios de búsqueda, como veremos en la sección 5.3.

Ahora estamos listos para ejecutar `fortune`:

```
$ fortune
Like so many Americans, she was trying to construct a life that made
sense from things she found in gift shops.
                -- Kurt Vonnegut, Jr.
```

Para el manejo de las variables de entorno, necesitamos saber un poco de la sintáxis empleada en su definición. Habrá situaciones en que necesitaremos substituir el contenido de una variable dentro de una cadena o rodeado por algun tipo de carácteres:

```
$ prefijo=al
$ echo ${prefijo}go
algo
$ echo cab${prefijo}go
cabalgo
```

o quizá sea necesario utilizar el valor de una variable previniendo el caso en que ésta no haya sido definida. Digamos que necesitamos almacenar en una variable el nombre de un archivo, pero que cuando este archivo no esté definido use por omisión el archivo salida. Para este caso tenemos la construcción `${variable-valor_alterno}`:

```
$ echo ${donde-salida}
salida
$ donde=aqui
$ echo ${donde-salida}
aqui
```

Variable	Significado dentro del *subshell*
`$#`	Número de argumentos recibidos
`$*`	Todos los argumentos
`$-`	Opciones dadas
`$?`	Valor de regreso del último comando ejecutado
`$$`	Identificador del proceso actual
`$HOME`	Directorio original del usuario
`$IFS`	Lista de carácteres que se utilizan como separadores en los argumentos
`$MAIL`	Archivo donde se almacena el correo electrónico del usuario
`$PATH`	Lista de directorios donde se buscan los comandos dados
`$PS1`	Prompt común del usuario
`$PS2`	Prompt cuando continúa la línea

Tabla 5.1: Variables más usuales

Como podemos ver en la primera línea imprimimos el valor de la variable `donde` si está definida o si no el valor por omisión `salida`, que es el que obtenemos en la segunda línea. Después de definir la variable `donde` en la tercera línea, al preguntar por ésta en la cuarta línea, obtenemos su definición en la quinta línea.

Existe también la opción de que en caso de que a la variable se le *asigne* el valor por omisión en caso de que no esté definida, con la construcción `${variable=valor_alterno}`:

```
$ echo ${donde=salida}
salida
$ echo $donde
salida
```

Ejercicio: Dé una construcción alterna de éste ejemplo.

Instructor: Se puede hacer: `donde=${donde-salida}` .

También se pueden utilizar la construcción `${variable?mensaje}`, donde si la variable no está definida, imprime el mensaje dado y termina la ejecución del *shell* en que fue invocada esta forma, o la forma alternativa `${variable+accion}` en la cual si la variable está definida, ejecuta la acción indicada, de otra forma no hace nada.

La tabla 5.1 muestra las variables usuales del *shell* y su función.

La utilidad de las variables la veremos con mayor detenimiento en la sección 5.1 donde construiremos algunos ejemplos interesantes. De momento, introduciremos la noción de *subshell* y *ejecución de archivos de comandos.*

En la sección de antecedentes mencionamos hasta el cansancio que lo que hace realmente poderoso a Unix es la filosofía con la que fue diseñado, es decir con la idea de que una colección de pequeños programas fuesen capaces de ser unidos por herramientas del sistema operativo con la finalidad de crear herramientas mayores o simplemente nuevas.

Introduciremos ahora los elementos para poder hacer esto. Como recordaremos, cualquier archivo con instrucciones puede convertirse en un archivo ejecutable con la instrucción `chmod`. Ahora veremos un pequeño ejemplo de un programa ejecutable. Como todavía no sabemos como añadir permanentemente a nuestro ambiente el *path* donde se encuentran los juegos, así que construiremos un pequeño programa que llamaremos `dichos` que añada el *path* y que ejecute el programa `fortune`. Al hacer ésto, lo que hacemos es pedirle al *shell* que ejecute una copia de sí mismo donde interpretará cada una de las instrucciones que le pedimos.

La secuencia siguiente es una de las que podemos utilizar para conseguir éste propósito:

```
$ cat > dichos
PATH=$PATH:/usr/games
fortune
^D
$ ls -al dichos
-rw-rw-rw-   1 pablo    usuarios       30 Jun 13 02:46 dichos
$ chmod ugo+rx dichos
$ ls -al dichos
-rwxr-xr-x   1 pablo    usuarios       30 Jun 13 02:46 dichos
$ dichos
According to the latest official figures, 43% of all statistics are
totally worthless.
$ echo $PATH
/bin:/usr/bin
```

Ejercicio: En el archivo `dichos` claramente estamos alterando el *path*. Pero en cuanto termina la ejecución y examinamos la variable, desaparece el directorio `/usr/games`. ¿Puede explicar qué pasó?

Instructor: Explicar que como claramente se dijo, al ejecutar un archivo con instrucciones, éstas son pasadas a una nueva copia del *shell*, cualquier alteración que hagamos, sólo tiene efecto en ésta copia del *shell*.

Regresemos ahora a la tabla 5.1. Aparece una palabra nueva en ella en la descripción de las variables `PS1` y `PS2`: *prompt*, hasta ahora no la habíamos mencionado, aún cuando ya estamos familiarizados con su significado. Simplemente es la manera que tiene el *shell* de avisarnos que está listo para la siguiente instrucción. En los ejemplos que hemos visto hasta ahora el *prompt* ha estado indicado por `$`, pero podemos cambiarlo por lo que queramos alterando éstas variables. Podemos cambiarla, por ejemplo, por:

```
$ PS1='Diga usted, amo: '
Diga usted, amo: date
Tue Jun 13 03:08:31 CST 1995
Diga usted, amo: PS1='Listo para la otra: '
Listo para la otra: PS1='Ordene, maestro del universo: '
Ordene, maestro del universo: PS1='(\u@\h)\w:[\d]:\t$ '
(pablo@breogan)~/trabajo/programas:[Mon Apr 10]:03:55:45$
```

y un largo etcétera de posibilidades y adulaciones al ego. En el último caso, Pablo decide que quiere poner en su prompt el nombre del usuario, la máquina en la que esta conectado —muy útil cuando se trabaja en una red grande y desde cuentas diferentes—, el directorio donde se encuentra y la fecha y hora.

5.3 Configuración del *shell*

Cuando el usuario entra en sesión, lo primero que ocurre es que el *shell* lee la configuración que el administrador definió para todo el sistema y después lee la del usuario en particular. Se encuentra en el archivo `.profile` en el directorio de trabajo del usuario. En él se guarda típicamente las trayectorias necesarias, el prompt que se utiliza y la ejecución de algunos programas iniciales. Este ejemplo es de un archivo de configuración muy escueto:

```
PATH=$PATH:$HOME/bin:/usr/games
PS1='\u@\h:\w$ '
date
who
```

5.4 Expresiones regulares

Se dice expresión regular de la descripción de una cadena en términos de repeticiones de carácteres. Así, por ejemplo, podemos decir que "palabra formada de una mayúscula seguida de una o más minúsculas" es la expresión regular de nombres. En términos de lenguaje matématico escribiríamos: $[A-Z][a-z]+$ con lo cual significamos lo mismo que en la frase anterior. Los carácteres para denotar la repetición son: `?` cero o una aparición, `*` cero o varias apariciones y `+` una o varias apariciones. Empleamos los corchetes `[]` para agrupar conjuntos de carácteres, en el ejemplo hablamos del conjunto de letras mayúsculas y en el segundo de las minúsculas. También podemos emplear el conjunto que no contiene a los carácteres dados, por ejemplo para hablar del conjunto de todos los carácteres salvo los signos de puntuación, diríamos: `[^,;\.]`, que como vemos, al punto lo precedemos de una diagonal invertida para *escapar*[2] su significado normal: cualquier carácter. El carácter `^` además tiene el significado de servir como ancla de principio de línea, mientras que el signo `$` sirve como ancla del fin de línea. Por ejemplo si queremos hablar de las líneas que comiencen con un signo de igual y que terminen con un punto, diríamos: `^=(.*)\.$`. Introducimos los paréntesis para especificar que después del siigno de igual se puede repetir cualquier carácter, aunque en el ejemplo es implícito.

Patrones repetitivos se pueden agrupar con paréntesis, por ejemplo para denotar la gramática que comprende a las palabras que comienzan con c y después se repite un numero arbitrario de veces la cadena ab y terminan con d, diríamos: `c(ab)*d`. Son palabras válidas en éste lenguaje: $cd, cabd, cababd, cabababd$, etc.

Cualquiera de los carácteres especiales que sea necesario utilizar en la expresión con su representación, basta con escaparlos con la diagonal invertida. Si queremos representar el conjunto de palabras —incluidos signos

[2]Decimos escapar significando que le quitamos su cambiamos su sentido original por el de un carácter más.

de puntuación, dígitos, etc— que van encerrados entre paréntesis, podemos usar: `( )(\(.*\))( ))`.

5.5 Estructuras de control

Para poder construir programas más complejos, necesitaremos estructuras de control que nos permitan cuestionar y tomar decisiones, controlar la repetición de ciclos y, por último, interaccionar con el usuario.

El *shell* nos permite esto de una manera eficiente, de forma que realmente podemos construir programas de gran funcionalidad sin recurrir a un lenguaje de programación de alto nivel, salvo para tareas muy específicas.

5.5.1 Condicionales

`if-then-else`

Volviendo un poco a nuestro ejemplo del corrector ortográfico para pobres[3], ¿qué pasa cuando corremos `co` si no existe el diccionario? Pues que truena espantosamente:

```
comm: diccionario: No such file or directory
```

Es claro que debemos de tener una manera de saber de antemano si el diccionario existe y avisar nosotros de éste error o tomar una acción adecuada. Bueno, el *shell* nos proporciona con un mecanismo para hacer preguntas y tomar decisiones y además para averiguar cosas acerca de archivos. Vayamos por partes.

Para preguntar y tomar decisiones tenemos, como en los lenguajes de programación para gente grande, la estructura de control *if-then-else* que nos permite examinar una condición y tomar una acción en caso de ser cierta u otra en caso de ser falsa. La sintáxis correcta es la siguiente:

```
if condicion
then
        accion_condicion_verdadera_1
        accion_condicion_verdadera_2
```

[3]©Doña Mechita Software Co. Inc. Marca Registrada

```
        ...
else
        accion_condicion_falsa_1
        accion_condicion_falsa_2
        ...
fi
```

Al igual que en los lenguajes de alto nivel, la parte del *else* puede no ir.

Para probar el valor de verdad de la parte condicionante tenemos el comando `test`. Para el ejemplo que estamos desarrollando, de momento nos basta con saber que podemos preguntar por la existencia de un archivo con el modificador `-e` de `test`. También podemos preguntar si el tamaño del archivo es de más de 0 carácteres, ya que no nos basta con saber que existe, sino que además debemos de garantizar que hay palabras en el diccionario. Esto lo podemos ver con el modificador `-s`. Además, como somos bien hechos, queremos garantizar que el archivo es legible para el usuario ejecutando nuestro fabuloso programa, y lo podemos saber con la opción `-r`. Ahora bien, estamos hablando de tres preguntas, `test` nos permite también pegarlas con conjunciones lógicas `y` y `o` con `-a` y `-o` respectivamente. Para este ejemplo necesitamos que las tres condiciones se cumplan, así que utilizaremos dos conjunciones `y`. Así, luego de todas estas provisiones, podemos aumentar nuestro programa `co` de la siguiente manera:

```
if test -e diccionario -a -s diccionario -a -r diccionario
then
        cat $1 | tr -c 'a-zA-Z\341\351\355\363\372\361' ' ' | \
        tr ' \t' '\n' | sort | uniq | comm -23 - diccionario | more
else
        echo "No existe el diccionario, o no lo puedo leer, o no"
        echo "contiene palabras."
        echo "Tomese dos aspirinas y llameme si persisten las molestias."
fi
```

Ahora ya tenemos una solución para cuando hay problemas con el diccionario. No es muy buena, pero es una solución. Incluso hay gente que se gana la vida dando respuestas como éstas.

En la tabla 5.2 podemos ver una lista de las opciones de `test`.

Modificador	Significado
`-d nombre`	Verdadero si el archivo existe y es un directorio
`-e nombre`	Verdadero si el archivo existe
`-f nombre`	Verdadero si el archivo existe y es un archivo regular
`-L nombre`	Verdadero si el archivo existe y es una liga simbólica
`-r nombre`	Verdadero si el archivo existe y es legible
`-s nombre`	Verdadero si el archivo existe y no está vacio
`-w nombre`	Verdadero si el archivo existe y es escribible por el usuario
`-x nombre`	Verdadero si el archivo existe y es ejecutable
`arch1 -nt arch2`	Verdadero si arch1 es más reciente que arch2
`arch1 -ot arch2`	Verdadero si arch1 es más viejo que arch2
`-z cadena`	Verdadero si la longitud de la cadena es cero
`-n cadena`	Verdadero si la longitud de la cadena es mayoor que cero
`cadena1 = cadena2`	Verdadero si las cadenas son iguales
`cadena1 != cadena2`	Verdadero si las cadenas son distintas
`! expr`	Verdadero si la expresión es falsa
`expr1 -a expr2`	Verdadero si ambas expresiones son verdaderas
`expr1 -o expr2`	Verdadero si alguna de las expresiones es verdadera
	Expresiones aritméticas, `arg1` y `arg2` son números:
`arg1 -eq arg2`	Verdadero si son iguales
`arg1 -ne arg2`	Verdadero si son diferentes
`arg1 -lt arg2`	Verdadero si el primero es menor
`arg1 -le arg2`	Verdadero si el primero es menor o igual
`arg1 -gt arg2`	Verdadero si el primero es mayor
`arg1 -ge arg2`	Verdadero si el primero es mayor o igual

Tabla 5.2: Modificadores de `test`. En el caso de las expresiones aritméticas, el modificador `-l` precediendo a una cadena se refiere a su longitud.

case

Aún cuando podemos anidar varios `if` para construir árboles de decisión más complejos, también tenemos la construcción `case-in-esac` para realizar acciones de acuerdo a un patrón dado:

```
echo "de que humor andas hoy?"
read humor
case $humor in
    bueno)  echo "Pues buenos dias! :-)" ;;
    malo)   echo "No me digas. :-(" ;;
    payaso) echo "No me hagas reir! :*)" ;;
    *)      echo "No se de que me hablas" ;;
esac
```

5.5.2 Ciclos

Tenemos dos tipos de ciclos de acuerdo al argumento, el `for`, que actúa sobre una lista de argumentos, y el `while` y `until`, que cotejan el valor de verdad del argumento.

for

La sintáxis es:

```
for variable in lista_de_valores
do
        accion_1
        accion_2
        ...
done
```

donde `variable` es una variable que toma cada uno de los valores en la lista dentro del cuerpo del ciclo. Por ejemplo:

```
for a in 1 2 3 lola lulu
do
        echo "cucamonga $i"
```

```
done
cucamonga 1
cucamonga 2
cucamonga 3
cucamonga lola
cucamonga lulu
```

Una aplicación típica es efecctuar una acción sobre una lista construída dinámicamente. Por ejemplo:

```
for i in `who | cut -d ' ' -f 1 | sort | uniq | tr '\n' ' '`
do
        write $1 << FIN
        Ya llegue!!!!
FIN
done
```

Con este pequeño programa le avisamos a todos los usuarios del sistema que ya entramos en sesión. Aclaremos: `who` nos dice quién está en sesión en ese momento, que además puede estar en varias sesiones simultáneas; `cut` nos da determinadas columnas de una columna o bien, como en éste caso de determinados campos utilizando las opciones `-d` para indicar el delimitador y `-f` para indicar que campo; con `sort` y `uniq` los ordenamos y unificamos como en el caso del corrector ortográfico; con `tr` cambiamos de uno por línea a todos en la misma línea (a la inversa que en el corrector) y con `write` enviamos el mensaje a la terminal del usuario.

La forma `<< delimitador` nos es completamente nueva. Con ella le decimos al *shell* que copie como entrada todo lo que se encuentre hasta que la palabra que utilizamos como delimitador aparezca por si sola en una línea.

`while` y `until`

La sintáxis del `while` es la siguiente:

```
while condicion
do
        accion_1
```

```
        accion_2
        ...
done
```

En este caso, las acciones se efectúan hasta que la condición se haga falsa.

La sintáxis para el `until` es igual, pero en este caso la acción se repite mientras la condición sea falsa:

```
until condicion
do
        accion_1
        accion_2
        ...
done
```

Construiremos el mismo ejemplo utilizando ambas formas. Se trata de un pequeño programa que nos avisa cuando un usuario entra en sesión. Además nos aseguraremos de que se le pase el argumento, avisando con un mensaje amigable cuando no sea así:

```
case $# in
    0) echo "Baboso! falta el nombre del usuario" ; exit 1 ;;
    1) usuario = $1 ;;
esac

while sleep 60
do
        who | grep $usuario
done
```

`sleep` es un programa que no hace nada durante el tiempo en segundos que se le indica. En este ejemplo, `sleep` duerme durante un minuto entre cada vez que el pregunta si el usuario está en sesión. Ahora con `until`:

```
case $# in
    0) echo "Baboso! falta el nombre del usuario" ; exit 1 ;;
    1) usuario = $1 ;;
esac
```

```
until who | grep $usuario
do
    sleep 60
done
```

Ejercicio: ¿Puede decir cuál es la diferencia primordial entre ambas versiones?

Instructor: En el caso del `while`, el programa espera un minuto antes de preguntar por primera vez si el usuario está en sesión, mientras que en la versión con `until` primero se pregunta y después se va a dormir.